

**IN THE UNITED STATES PATENT AND TRADEMARK OFFICE**

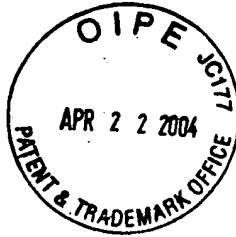
In re Patent Application of

WATT et al

Serial No. 10/714,561

Filed: November 17, 2003

For: CONTROL OF ACCESS TO A MEMORY BY A DEVICE



Atty. Ref.: 550-486

TC/A.U.: 2131

Examiner:

\* \* \* \* \*

April 22, 2004

Commissioner for Patents  
P.O. Box 1450  
Alexandria, VA 22313-1450

Sir:

**SUBMISSION OF PRIORITY DOCUMENTS**

It is respectfully requested that this application be given the benefit of the foreign filing date under the provisions of 35 U.S.C. §119 of the following, a certified copy of which is submitted herewith:

<u>Application No.</u>	<u>Country of Origin</u>	<u>Filed</u>
0226875.3	UK	18 November 2002
0226879.5	UK	18 November 2002
0303446.9	UK	14 February 2003

Respectfully submitted,

**NIXON & VANDERHYE P.C.**

By: \_\_\_\_\_

Stanley C. Spooner  
Reg. No. 27,393

SCS:kmm  
1100 North Glebe Road, 8th Floor  
Arlington, VA 22201-4714  
Telephone: (703) 816-4000  
Facsimile: (703) 816-4100



INVESTOR IN PEOPLE

The Patent Office  
Concept House  
Cardiff Road  
Newport  
South Wales  
NP10 8QQ

I, the undersigned, being an officer duly authorised in accordance with Section 74(1) and (4) of the Deregulation & Contracting Out Act 1994, to sign and issue certificates on behalf of the Comptroller-General, hereby certify that annexed hereto is a true copy of the documents as originally filed in connection with the patent application identified therein.

In accordance with the Patents (Companies Re-registration) Rules 1982, if a company named in this certificate and any accompanying documents has re-registered under the Companies Act 1980 with the same name as that with which it was registered immediately before re-registration save for the substitution as, or inclusion as, the last part of the name of the words "public limited company" or their equivalents in Welsh, references to the name of the company in this certificate and any accompanying documents shall be treated as references to the name with which it is so re-registered.

In accordance with the rules, the words "public limited company" may be replaced by p.l.c., plc, P.L.C. or PLC.

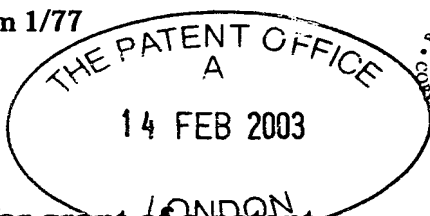
Re-registration under the Companies Act does not constitute a new legal entity but merely subjects the company to certain additional company law rules.

Signed

Dated 4 November 2003

Patents Form 1/77

Patents Act 1977  
(Rule 16)



17FEB03 E785162-1 002246  
P01/7700 0.00-0303446.9

**Request for grant of a patent**

(See the notes on the back of this form. You can also get an explanatory leaflet from the Patent Office to help you fill in this form)

The Patent Office

Cardiff Road  
Newport  
South Wales  
NP10 8QQ

1. Your reference	P015748GB		
2. Patent application number (The Patent Office will fill in this part)	0303446.9		
3. Full name, address and postcode of the or of each applicant (underline all surnames)	ARM Limited 110 Fulbourn Road Cherry Hinton Cambridge CB1 9NJ  Patents ADP number (if you know it)  If the applicant is a corporate body, give the country/state of its incorporation		
	United Kingdom  7498124002		
4. Title of the invention	APPARATUS AND METHOD FOR CONTROLLING ACCESS TO A MEMORY		
5. Name of your agent (if you have one)	D Young & Co		
	"Address for service" in the United Kingdom to which all correspondence should be sent (including the postcode)  21 New Fetter Lane London EC4A 1DA  Patents ADP number (if you know it)		
	59006		
6. If you are declaring priority from one or more earlier patent applications, give the country and the date of filing of the or of each of these earlier applications and (if you know it) the or each application number	Country	Priority application number (if you know it)	Date of filing (day / month / year)
	See Attached Schedule		
7. If this application is divided or otherwise derived from an earlier UK application, give the number and the filing date of the earlier application	Number of earlier application	Date of filing (day / month / year)	
8. Is a statement of inventorship and of right to grant of a patent required in support of this request? (Answer 'Yes' if: a) any applicant named in part 3 is not an inventor, or b) there is an inventor who is not named as an applicant, or c) any named applicant is a corporate body. See note (d))	Yes		

## SCHEDULE OF PRIORITIES CLAIMED

<u>Country</u>	<u>Priority Application Number</u>	<u>Date of Filing</u>
United Kingdom	0226886.0	18 November 2002
United Kingdom	0226875.3	18 November 2002
United Kingdom	0226882.9	18 November 2002
United Kingdom	0226879.5	18 November 2002
United Kingdom	0226890.2	18 November 2002
United Kingdom	0226906.6	18 November 2002
United Kingdom	0226902.5	18 November 2002
United Kingdom	0226884.5	18 November 2002
United Kingdom	0226874.6	18 November 2002
United Kingdom	0226905.8	18 November 2002
United Kingdom	0226908.2	18 November 2002
United Kingdom	0226912.4	18 November 2002
United Kingdom	0226887.8	18 November 2002
United Kingdom	0226888.6	18 November 2002

DUPLICATE

P015748GB

1

APPARATUS AND METHOD FOR CONTROLLING  
ACCESS TO A MEMORY

5 The present invention relates to an apparatus and method for controlling access to a memory.

A data processing apparatus will typically include a processor for running applications loaded onto the data processing apparatus. The processor will operate under the control of an operating system. The data required to run any particular application will typically be stored within a memory of the data processing apparatus. It  
10 will be appreciated that the data may consist of the instructions contained within the application and/or the actual data values used during the execution of those instructions on the processor.

There arise many instances where the data used by at least one of the applications is sensitive data that should not be accessible by other applications that can  
15 be run on the processor. An example would be where the data processing apparatus is a smart card, and one of the applications is a security application which uses sensitive data, such as for example secure keys, to perform validation, authentication, decryption and the like. It is clearly important in such situations to ensure that such sensitive data is kept secure so that it cannot be accessed by other applications that may be loaded onto  
20 the data processing apparatus, for example hacking applications that have been loaded onto the data processing apparatus with the purpose of seeking to access that secure data.

In known systems, it has typically been the job of the operating system developer to ensure that the operating system provides sufficient security to ensure that the secure  
25 data of one application cannot be accessed by other applications running under the control of the operating system. However, as systems become more complex, the general trend is for operating systems to become larger and more complex, and in such situations it becomes increasingly difficult to ensure sufficient security within the operating system itself.

30 Examples of systems seeking to provide secure storage of sensitive data and to provide protection against malicious program code are those described in United States

Patent Application US 2002/0007456 A1 and United States Patents US 6,282,657 B and US 6,292,874 B.

Accordingly, it is desirable to provide an improved technique for seeking to retain the security of such secure data contained within the memory of the data processing apparatus.

Viewed from a first aspect, the present invention provides a data processing apparatus, comprising: a processor operable in a plurality of modes and a plurality of domains, said plurality of domains comprising a secure domain and a non-secure domain, said plurality of modes including at least one non-secure mode being a mode in the non-secure domain, and at least one secure mode being a mode in the secure domain, said processor being operable such that when executing a program in a secure mode said program has access to secure data which is not accessible when said processor is operating in a non-secure mode; a memory operable to store data required by the processor and comprising secure memory for storing secure data and non-secure memory for storing non-secure data, the processor being operable to issue a memory access request when access to an item of data in the memory is required; at least one memory management unit operable, upon receipt of the memory access request from the processor, to perform conversion of a virtual address specified by the memory access request to a physical address; a first set of tables, each table in the first set containing a number of first descriptors, each first descriptor containing at least a virtual address portion and a corresponding intermediate address portion; a second set of tables, each table in the second set containing a number of second descriptors, each second descriptor containing at least an intermediate address portion and a corresponding physical address portion, the second set of tables being managed by the processor when operating in a privileged mode which is not a non-secure mode; the at least one memory management unit being operable to cause predetermined tables in said first and second set to be referenced to enable the conversion of the virtual address specified by the memory access request to a physical address.

In accordance with the present invention, the processor is arranged to be operable in a plurality of modes and a plurality of domains, including at least one non-secure mode being a mode in the non-secure domain, and at least one secure mode being a mode in the secure domain. Further, a memory is provided for storing data required by

physical address portion. The second set of tables are managed by the processor when operating in a privileged mode which is not a non-secure mode, and hence can be considered as secure. By this approach, the operating system running in either domain sees as its physical address space an intermediate address space, and the intermediate  
5 address space can be defined for each domain via the second set of tables. This hence allows complete control of the environment of the non-secure operating system used in the non-secure domain, since the secure memory regions can be completely hidden from the non-secure operating system's view of its "physical address" space, since it sees as its physical address space the intermediate address space, which can be arranged to have  
10 a contiguous sequence of intermediate addresses.

A further problem exhibited if only a single set of tables is used, rather than the first and second sets envisaged by the present invention, is that, given the different views of memory between the non-secure domain and the secure domain, the physical address space for the memory accessible in one domain can start at zero, but this will not  
15 typically be the case for the other domain. In that domain, the fact that the physical address space does not start at zero will be evident to the operating system being used in that domain, and this would hence require that operating system to have a certain knowledge about the other domain. However, use of the present invention gives the operating system in either domain the ability to see its own memory space starting from  
20 zero, which is what an operating system would typically expect.

Hence, in accordance with the present invention, when the at least one memory management unit needs to determine a physical address for a virtual address specified by a memory access request, it will cause predetermined tables in the first and second set to be referenced to enable the conversion of the virtual address to a physical address. The  
25 use of such a two-level memory protection technique enables the complexity introduced by the segmentation of the physical address space into secure memory and non-secure memory to be shielded from the operating systems running in each domain, and thereby enables the physical address space to be segmented as desired.

This technique also facilitates easier manipulation of memory regions within  
30 the physical address space, since it is now possible to swap a region of memory between non-secure memory and secure memory merely by manipulation of the relevant second descriptor(s) in the second set of tables.

In one embodiment, the at least one memory management unit comprises a first memory management unit and a second memory management unit. In one particular embodiment, the tables in the first set are associated with the first memory management unit and the tables in the second set are associated with the second memory management unit.

In such embodiments, if the first memory management unit needs to access a first descriptor within a predetermined table of said first set, it issues a table lookup request specifying an intermediate address for that first descriptor, the second memory management unit being operable to receive the table lookup request and determine the physical address corresponding to that intermediate address. Hence, the intermediate address generated by the first memory management unit is passed to the second memory management unit for resolving into a physical address.

Preferably, the second memory management unit is then operable to cause the first descriptor at that physical address to be retrieved and returned to the first memory management unit.

In one embodiment, the first memory management unit comprises a first internal storage unit for storing first descriptors retrieved from the predetermined table of the first set, and used by the first memory management unit to derive access control information used to perform the conversion of the virtual address into a corresponding intermediate address.

It will be appreciated that the first internal storage unit can take a variety of forms. However, in one embodiment, the first internal storage unit is a first translation lookaside buffer (TLB) operable to store the first descriptors retrieved from the predetermined table of the first set.

It will be appreciated that in one embodiment, the first internal storage may solely comprise the above first TLB. However, in one particular embodiment, the first TLB is a first main TLB for storing the first descriptors retrieved by the first memory management unit from the predetermined table of the first set, and the internal storage further comprises a micro-TLB for storing the access control information derived from the first descriptors, the access control information comprising conversions between a number of virtual address portions and corresponding intermediate address portions, and the access control information being transferred from the first main TLB to the micro-



micro-TLB for storing the access control information derived from the second descriptors, the access control information comprising conversions between a number of intermediate address portions and corresponding physical address portions, and the access control information being transferred from the second main TLB to the micro-TLB prior to use of that access control information by the second memory management unit.

In preferred embodiments, the first and second sets of tables each comprise at least a secure table and a non-secure table, the first and second TLBs comprising a flag associated with each descriptor stored therein to identify whether that descriptor is derived from said non-secure table or said secure table.

By the use of such a flag, the main TLB within each MMU does not need to be flushed every time the processor switches between a secure mode and a non-secure mode, or vice versa, and accordingly performance is not adversely affected as a result of the provision of the non-secure table and the secure table. This is advantageous whether or not the internal storage further includes a micro-TLB.

However, in embodiments where a micro-TLB is also provided, the micro-TLB of both the first and second memory management units is preferably flushed whenever the mode of operation of the processor changes between a secure mode and a non-secure mode, in the secure mode access control information only being transferred to the micro-TLB from a descriptor in the associated first or second main TLB that said associated flag indicates is from the secure table, and in the non-secure mode access control information only being transferred to the micro-TLB from a descriptor in the associated first or second main TLB that said associated flag indicates is from the non-secure table. Clearly if only one of the memory management units contains a micro-TLB, then only that memory management unit will be subjected to such a flushing operation when the mode of operation of the processor changes between a secure mode and a non-secure mode.

Since the memory management units reference their respective micro-TLBs when performing the conversion of the virtual address specified by the memory access request into a physical address, this approach ensures that only information derived from descriptors in the non-secure tables can be used to perform such conversion when the processor is operating in a non-secure mode.

intermediate address, whereafter the table merging code is operable to merge the first and second descriptors to produce the new descriptor.

It will be appreciated that the single memory management unit of such embodiments may take a variety of forms. However, in one embodiment, the single  
5 memory management unit comprises an internal storage unit for storing the new descriptor produced by the table merging code, and used by the single memory management unit to derive access control information used to perform the conversion of the virtual address into a corresponding physical address.

Preferably, the processor is operable to execute the table merging code when the  
10 access control information required to determine the physical address for the memory access request is not found in the internal storage unit.

The internal storage unit may take a variety of forms. However, in one embodiment the internal storage unit is a translation lookaside buffer (TLB) operable to store the new descriptors produced by the table merging code.

More particularly, in one embodiment the TLB is a main TLB for storing the  
15 new descriptors obtained by the single memory management unit from the table merging code, and the internal storage further comprises a micro-TLB for storing the access control information derived from the new descriptors, the access control information comprising conversions between a number of virtual address portions and corresponding  
20 physical address portions, and the access control information being transferred from the main TLB to the micro-TLB prior to use of that access control information by the single memory management unit.

In one embodiment, the first and second sets of tables each comprise at least a secure table and a non-secure table, the TLB comprising a flag associated with each new  
25 descriptor stored therein to identify whether that new descriptor is derived from said non-secure tables or said secure tables.

In such embodiments, the micro-TLB of the single memory management unit is preferably flushed whenever the mode of operation of the processor changes between a secure mode and a non-secure mode, in the secure mode access control information  
30 only being transferred to the micro-TLB from a new descriptor in the main TLB that said associated flag indicates is derived from secure tables, and in the non-secure mode access control information only being transferred to the micro-TLB from a new

apparatus comprising a processor operable in a plurality of modes and a plurality of domains, said plurality of domains comprising a secure domain and a non-secure domain, said plurality of modes including at least one non-secure mode being a mode in the non-secure domain, and at least one secure mode being a mode in the secure domain, said processor being operable such that when executing a program in a secure mode said program has access to secure data which is not accessible when said processor is operating in a non-secure mode, the memory being operable to store data required by the processor and comprising secure memory for storing secure data and non-secure memory for storing non-secure data, the method comprising the steps of:

10 providing a first set of tables, each table in the first set containing a number of first descriptors, each first descriptor containing at least a virtual address portion and a corresponding intermediate address portion; providing a second set of tables, each table in the second set containing a number of second descriptors, each second descriptor containing at least an intermediate address portion and a corresponding physical address

15 portion, the second set of tables being managed by the processor when operating in a privileged mode which is not a non-secure mode; issuing from the processor a memory access request when access to an item of data in the memory is required; and performing conversion of a virtual address specified by the memory access request to a physical address with reference to predetermined tables in said first and second set.

20 Viewed from a third aspect, the present invention provides a computer program providing table merging code and operable to configure a processor of a data processing apparatus to perform the step of: referencing the predetermined tables of the first and second sets in order to produce from a first descriptor and an associated second descriptor a new descriptor associating a virtual address portion with a corresponding

25 physical address portion. A fourth aspect of the present invention provides a computer program product carrying such a computer program.

The present invention will be described further, by way of example only, with reference to preferred embodiments thereof as illustrated in the accompanying drawings, in which:

Figure 12 illustrates an alternative scheme for the handling of non-secure interrupt request signals and secure interrupt request signals compared to that illustrated in Figure 10;

5        Figures 13A and 13B illustrate example scenarios for dealing with a non-secure interrupt request and a secure interrupt request in accordance with the scheme illustrated in Figure 12;

Figure 14 is an example of a vector interrupt table;

10

Figure 15 schematically illustrates multiple vector interrupt tables associated with different security domains;

Figure 16 schematically illustrates an exception control register;

15

Figure 17 is a flow diagram illustrating how an instruction attempting to change a processing status register in a manner that alters the security domain setting can generate a separate mode change exception which in turn triggers entry into the monitor mode and running of the monitor program;

20

Figure 18 schematically shows a thread of control of a processor operating in a plurality of modes, wherein a task in monitor mode is interrupted;

Figure 19 schematically shows a different thread of control of a processor  
25    operating in a plurality of modes;

Figure 20 schematically shows a further thread of control of a processor operating in a plurality of modes, wherein interrupts are enabled in monitor mode;

30        Figures 21 to 23 illustrate a view of different processing modes and scenario for switching between secure and non-secure domains in accordance with another example embodiment(s);

Figure 35 illustrates how the processor configuration data is overridden with monitor mode specific processor configuration data when the processor is operating in monitor mode;

5

Figure 36 is a flow diagram illustrating how the processor configuration data is switched when transitioning between the secure domain and the non-secure domain in accordance with one embodiment to the present invention;

10

Figure 37 is a diagram illustrating the memory management logic used in one embodiment of the present invention to control access to memory;

Figure 38 is a block diagram illustrating the memory management logic of a second embodiment of the present invention used to control access to memory;

15

Figure 39 is a flow diagram illustrating the process performed in one embodiment of the present invention within the memory management logic to process a memory access request that specifies a virtual address;

20

Figure 40 is a flow diagram illustrating the process performed in one embodiment of the present invention within the memory management logic to process a memory access request that specifies a physical address;

25

Figure 41 schematically illustrates how the partition checker of preferred embodiments is operable to prevent access to a physical address within secure memory when the device issuing the memory access request is operating in a non-secure mode;

30

Figure 42 is a diagram illustrating the use of both a non-secure page table and a secure page table in preferred embodiments of the present invention;

Figure 51 illustrates, by way of example, the correspondence between physical address space and intermediate address space for both the secure domain and the non-secure domain;

5           Figure 52 illustrates the swapping of memory regions between secure and non-secure domains through manipulation of the page tables associated with the second MMU;

10           Figure 53 is an embodiment illustrating an implementation using a single MMU, and where a miss in the main TLB causes an exception to be invoked to determine the virtual to physical address translation;

15           Figure 54 is a flow diagram illustrating the process performed by the processor core in order to action an exception issued upon occurrence of a miss in the main TLB of the MMU of Figure 53;

20           Figure 55 is a block diagram illustrating components provided within a data processing apparatus of one embodiment, in which the cache is provided with information as to whether the data stored in individual cache lines is secure data or non-secure data;

            Figure 56 illustrates the construction of the memory management unit illustrated in Figure 55;

25           Figure 57 is a flow diagram illustrating the processing performed within the data processing apparatus of Figure 55 to process a non-secure memory access request;

            Figure 58 is a flow diagram illustrating the processing performed within the data processing apparatus of Figure 55 in order to process a secure memory access request;

30           Figure 59 schematically shows possible granularity of monitoring functions for different modes and applications running on a processor;

logic unit (ALU) 16 arranged to execute sequences of instructions. Data required by the ALU 16 is stored within a register bank 14. The core 10 is provided with various monitoring functions to enable diagnostic data to be captured indicative of the activities of the processor core. As an example, an Embedded Trace Module (ETM) 22 is provided for producing a real time trace of certain activities of the processor core in dependence on the contents of certain control registers 26 within the ETM 22 defining which activities are to be traced. The trace signals are typically output to a trace buffer from where they can subsequently be analysed. A vectored interrupt controller 21 is provided for managing the servicing of a plurality of interrupts which may be raised by various peripherals (not illustrated).

Further, as shown in Figure 1, another monitoring functionality that can be provided within the core 10 is a debug function, a debugging application external to the data processing apparatus being able to communicate with the core 10 via a Joint Test Access Group (JTAG) controller 18 which is coupled to one or more scan chains 12. Information about the status of various parts of the processor core 10 can be output via the scan chains 12 and the JTAG controller 18 to the external debugging application. An In Circuit Emulator (ICE) 20 is used to store within registers 24 conditions identifying when the debug functions should be started and stopped, and hence for example will be used to store breakpoints, watchpoints, etc.

The core 10 is coupled to a system bus 40 via memory management logic 30 which is arranged to manage memory access requests issued by the core 10 for access to locations in memory of the data processing apparatus. Certain parts of the memory may be embodied by memory units connected directly to the system bus 40, for example the Tightly Coupled Memory (TCM) 36, and the cache 38 illustrated in Figure 1. Additional devices may also be provided for accessing such memory, for example a Direct Memory Access (DMA) controller 32. Typically, various control registers 34 will be provided for defining certain control parameters of the various elements of the chip, these control registers also being referred to herein as coprocessor 15 (CP 15) registers.

direction. From a view external to the core the monitor mode is always secure and the monitor program is in secure memory.

Within the non-secure domain there is provided a non-secure operating system 74 and a plurality of non-secure application programs 76, 78 which execute in co-  
5 operation with the non-secure operating system 74. In the secure domain, a secure kernel program 80 is provided. The secure kernel program 80 can be considered to form a secure operating system. Typically such a secure kernel program 80 will be designed to provide only those functions which are essential to processing activities  
10 which must be provided in the secure domain such that the secure kernel 80 can be as small and simple as possible since this will tend to make it more secure. A plurality of secure applications 82, 84 are illustrated as executing in combination with the secure kernel 80.

15 Figure 3 illustrates a matrix of processing modes associated with different security domains. In this particular example the processing modes are symmetrical with respect to the security domain and accordingly Mode 1 and Mode 2 exist in both secure and non-secure forms.

20 The monitor mode has the highest level of security access in the system and is the only mode entitled to switch the system between the non-secure domain and the secure domain in either direction. Thus, all domain switches take place via a switch to the monitor mode and the execution of the monitor program 72 within the monitor mode.

25 Figure 4 schematically illustrates another set of non-secure domain processing modes 1, 2, 3, 4 and secure domain processing modes a, b, c. In contrast to the symmetric arrangement of Figure 3, Figure 4 shows that some of the processing modes may not be present in one or other of the security domains. The monitor mode  
30 86 is again illustrated as straddling the non-secure domain and the secure domain. The monitor mode 86 can be considered a secure processing mode, since the secure status flag may be changed in this mode and monitor program 72 in the monitor mode



secure domain to the secure domain as may be permitted and desirable by using the fast and efficient mechanism of placing it in a register which is accessible in both the non-secure domain and the secure domain.

5           An important advantage of having secure register bank is to avoid the need for flushing the contents of registers before switching from one world to the other. If latency is not a critical issue, a simpler hardware system with no duplicated registers for the secure domain world may be used, e.g. Figure 6. The monitor mode is responsible switching from one domain to the other. Restoring context, saving  
10 previous context, as well as flushing registers is performed by a monitor program at least partially executing in monitor mode. The system behaves thus like a virtualisation model. This type of embodiment is discussed further below. Reference should be made to, for example, the programmer's model of the ARM7 upon which the security features described herein build.

15

### *Processor Modes*

Instead of duplicating modes in secure world, the same modes support both secure and non-secure domains (see Figure 8). Monitor mode is aware of the current  
20 status of the core, either secure or non-secure (e.g. as read from an S bit stored in a coprocessor configuration register).

In the Figure 8, whenever an SMI (Software Monitor Interrupt instruction) occurs, the core enters monitor mode to switch properly from one world to the other.

25

With reference to Figure 9 in which SMIs are permitted from user mode:

1. The scheduler launches thread 1
2. Thread 1 needs to perform a secure function => SMI secure call, the core enters  
30 monitor mode. Under hardware control the current PC and CPSR (current processor status register) are stored in R14\_mon and SPSR\_mon (saved processor status register for the monitor mode) and IRQ/FIQ interrupts are disabled.

When a hardware reset occurs, the MMU is disabled and the ARM core (processor) branches to secure supervisor mode with the S bit set. Once the secure boot is terminated an SMI to go to monitor mode may be executed and the monitor can switch to the OS in non-secure world (non-secure svc mode) if desired. If it is desired to use a legacy OS this can simply boot in secure supervisor mode and ignore the secure state.

### **SMI INSTRUCTION**

This instruction (a mode switching software interrupt instruction) can be called from any non-secure modes in the non-secure domain (as previously mentioned it may be desired to restrict SMIs to privileged modes), but the target entry point determined by the associated vector is always fixed and within monitor mode. Its up to the SMI handler to branch to the proper secure function that must be run (e.g. controlled by an operand passed with the instruction).

Passing parameters from non-secure world to secure world can be performed using the shared registers of the register bank within a Figure 6 type register bank.

When a SMI occurs in non-secure world, the ARM core may do the following actions in hardware:

- Branch to SMI vector (in secure memory access is allowed since you will now be in monitor mode) into monitor mode
- Save PC into R14\_mon and CPSR into SPSR\_mon
- Set the S bit using the monitor program
- Start to execute secure exception handler in monitor mode (restore/save context in case of multi-threads)
- Branch to secure user mode (or another mode, like svc mode) to execute the appropriate function
- IRQ and FIQ are disabled while the core is in monitor mode (latency is increased)

### **Secure World Exit**

**Solution One**

**In this solution, two distinct pins are required to support secure and non-secure interrupts.**

- 5      While in Non Secure world, if
- an IRQ occurs, the core goes to IRQ mode to handle this interrupt as in ARM cores such as the ARM7
  - a SIRQ occurs, the core goes to monitor mode to save non-secure context and then to a secure IRQ handler to deal with the secure interrupt.

10

***While in Secure world, if***

- an SIRQ occurs, the core goes to the secure IRQ handler. The core does not leave the secure world
  - an IRQ occurs, the core goes to monitor mode where secure context is saved, then
- 15            to a non-secure IRQ handler to deal with this non-secure interrupt.

In other words, when an interrupt that does not belong to the current world occurs, the core goes directly to monitor mode, otherwise it stays in the current world (see Figure 10).

20

**IRQ Occurring in Secure World**

See Figure 11A:

- 25      1.    The scheduler launches thread 1.
2.    Thread 1 needs to perform a secure function => SMI secure call, the core enters monitor mode. Current PC and CPSR are stored in R14\_mon and SPSR\_mon, IRQ/FIQ are disabled.
3.    The monitor handler (program) does the following tasks:
- 30      -    The S bit is set.

### **SIRQ Occurring in Non-Secure World**

See Figure 11B:

1. The schedule launches thread 1
  - 5 2. A SIRQ occurs while secure thread 1 is running. The core jumps directly to monitor mode (specific vector) and stores current PC in R14\_mon and CPSR in SPSR\_mon in monitor mode, IRQ/FIQ are then disabled.
  3. Non-Secure context must be saved, then the core goes to a secure IRQ handler.
  4. The IRQ handler services the SIRQ, then gives control back to the monitor
  - 10 mode handler using an SMI with appropriate parameters.
  5. The monitor handler restores non-secure context so that a SUBS instruction makes the core return to the non-secure world and resumes the interrupted thread 1.
  6. Thread 1 executes until the end, then gives the hand back to the scheduler.
- 15 The mechanism of Figure 11A has the advantage of providing a deterministic way to enter secure world. However, there are some problems associated with interrupt priority: e.g. while a SIRQ is running in secure interrupt handler, a non-secure IRQ with higher priority may occur. Once the non-secure IRQ is finished, there is a need to recreate the SIRQ event so that the core can resume the secure
- 20 interrupt.

### **Solution Two**

In this mechanism (See Figure 12) two distinct pins, or only one, may support secure and non-secure interrupts. Having two pins reduces interrupt latency.

25

While in Non Secure world, if

- an IRQ occurs, the core goes to IRQ mode to handle this interrupt like in ARM7 systems
- a SIRQ occurs, the core goes to an IRQ handler where an SMI instruction will
- 30 make the core branch to monitor mode to save non-secure context and then to a secure IRQ handler to deal with the secure interrupt.

also save in these same registers the non-secure context that must be restored once the IRQ routine will be finished.

7. The IRQ handler services the IRQ, then gives control back to thread 1 in the non-secure world. By restoring `SPRS_irq` and `R14_irq` into the CPSR and PC, the core is now pointing onto the SMI instruction that has been interrupted.
8. The SMI instruction is re-executed (same instruction as 2).
9. The monitor handler sees this thread has previously been interrupted, and restores the thread 1 context. It then branches to secure thread 1 in user mode, pointing to the instruction that has been interrupted.
10. Secure thread 1 runs until it finishes, then branches onto the 'return from secure'; function in monitor mode (dedicated SMI).
11. The 'return from secure' function does the following tasks:
  - indicates that secure thread 1 is finished (i.e., in the case of a thread ID table, remove thread 1 from the table).
- 15 - restores from stack non-secure context and flushes required registers, so that no secure information can be read once we return in non-secure world.
  - branches back to the non-secure world with a SUBS instruction, restoring the PC (from restored `R14_mon`) and CPSR (from `SPSR_mon`). The return point in the non-secure world should be the instruction following the previously executed SMI in thread 1.
11. Thread 1 executes until the end, then gives the hand back to the scheduler.

### **SIRQ Occurring in Non-Secure World**

See Figure 13B:

25

1. The schedule launches thread 1.
2. A SIRQ occurs while secure thread 1 is running.
3. The core jumps directly irq mode and stores current PC in `R14_irq` and CPSR in `SPSR_irq`. IRQ is then disabled. The IRQ handler detects this is a SIRQ and performs a SMI instruction with appropriate parameters.
4. Once in monitor mode, non-secure context must be saved, then the core goes to a secure IRQ handler.

30

Prefetch Abort	0x0C	Abort mode/Monitor mode
Data Abort	0x10	Abort mode/Monitor Mode
IRQ/SIRQ	0x18	IRQ mode
FIQ	0x1X	FIQ mode
SMI	0x20	Undef mode/Monitor mode

NB. The Reset entry is only in the secure vector table. When a Reset is performed in non secure world, the core hardware forces entry of supervisor mode and setting of the S bit so that the Reset vector can be accessed in secure memory.

5

Figure 15 illustrates three exception vector tables respectively applicable to a secure mode, a non-secure mode and the monitor mode. These exception vector tables may be programmed with exception vectors in order to match the requirements and characteristics of the secure and non-secure operating systems. Each of the exception vector tables may have an associated vector table base address register within CP15 storing a base address pointing to that table within memory. When an exception occurs the hardware will reference the vector table base address register corresponding to the current state of the system to determine the base address of the vector table to be used. Alternatively, the different virtual to physical memory mappings applied in the different modes may be used to separate the three different vector table stored at different physical memory addresses. As illustrated in Figure 16, an exception trap mask register is provided in a system (configuration controlling) coprocessor (CP15) associated with the processor core. This exception trap mask register provides flags associated with respective exception types. These flags indicate whether the hardware should operate to direct processing to either the vector for the exception concerned within its current domain or should force a switch to the monitor mode (which is a type of secure mode) and then follow the vector in the monitor mode vector table. The exception trap mask register (exception control register) is only writable from the monitor mode. It may be that read access is also prevented to the exception trap mask register when in a non-secure mode. It will be

25

new type of exception. At step 98 the hardware detects any instruction which is attempting to change to monitor mode as indicated in a current program status register (CPSR). When such an attempt is detected, then a new type of exception is triggered, this being referred to herein as a CPSR violation exception. The generation of this  
5 CPSR violation exception at step 100 results in reference to an appropriate exception vector within the monitor mode and the monitor program is run at step 102 to handle the CPSR violation exception.

It will be appreciated that the mechanisms for initiating a switch between secure  
10 domain and non-secure domain discussed in relation to Figure 17 may be provided in addition to support for the SMI instruction previously discussed. This exception mechanism may be provided to respond to unauthorised attempts to switch mode as all authorised attempts should be made via an SMI instruction. Alternatively, such a mechanism may be legitimate ways to switch between the secure domain and the non-  
15 secure domain or may be provided in order to give backwards compatibility with existing code which, for example, might seek to clear the processing status register as part of its normal operation even though it was not truly trying to make an unauthorised attempt to switch between the secure domain and the non-secure domain.

20

As described above, in general interrupts are disabled when the processor is operating in monitor mode. This is done to increase the security of the system. When an interrupt occurs the state of the processor at that moment is stored in interrupt exception registers so that on completion of the interrupt function the processing of  
25 the interrupted function can be resumed at the interrupt point. If this process were allowed in monitor mode it could reduce the security of the monitor mode, giving a possible secure data leakage path. For this reason interrupts are generally disabled in monitor mode. However, one consequence of disabling interrupts during monitor mode is that interrupt latency is increased.

30

It would be possible to allow interrupts in monitor mode if the state of the processor executing the function was not stored. This can only be done if following

As is shown in Figure 18 following completion of the interrupt task, task B, the processor reads the address of the SMI instruction which has been copied to the interrupt register and performs an SMI and starts to process function C again.

5       The above process only works if function C is restartable, that is to say if restarting process C will result in repeatable processing steps. This will not be the case if function C has changed any of the states of the processor such as the stack pointer that may affect its future processing. A function that is repeatable in this way is said to have idempotence. One way of dealing with the problem of a function not  
10   having idempotence is to rearrange the code defining the function in such a way that the first portion of the code has idempotence and once it is no longer possible to arrange the code to have idempotence interrupts are disabled. For example, if code C involves writing to the stack, it may be possible to do so without updating the stack pointer at least at first. Once it is decided that the code can no longer feasibly be  
15   safely restarted, then the code for function C can instruct the processor to disable interrupts and then it can update the stack pointer to the correct position. This is shown in Figure 18 where interrupts are disabled a certain way through the processing of function C.

20       Figure 19 illustrates a slightly different example. In this example, a certain way through the processing of task C, a further control parameter is set. This indicates that the following portion of task C is not strictly idempotent, but can be safely restarted provided that a fix-up routine is run first. This fix-up routine acts to restore a state of the processor to how it was at the start of task C, such that task C can be safely  
25   restarted and produce the same processor state at the end of the task as it would have done had it not been interrupted. In some embodiments at the point that the further control parameter is set interrupts may be disabled for a short while while some states of the processor are amended such as the stack pointer being updated. This allows the processor to be restored to an idempotent state later.

30

When an interrupt occurs after the further control parameter has been set, then there are two possible ways to proceed. Either the fix-up routine can be performed



Although the above described way of dealing with interrupt latency has been described with respect to a system having secure and non-secure domains and a monitor mode, it is clearly applicable to any system which has functions that should not be resumed for a particular reason. Generally such functions operate by disabling interrupts which increase interrupt latency. Amending the functions to be restartable and controlling the processor to restart them following an interrupt allows the interrupts to be enabled for at least a portion of the processing of the function and helps reduce interrupt latency. For example normal context switching of an operating system.

#### Access to secure and non-secure memory

As described with reference to Figure 1, the data processing apparatus has memory, which includes, inter alia, the TCM 36, cache 38, ROM 44, memory of slave devices and external memory 56. As described with reference to Figure 37 for example, memory is partitioned into secure and non-secure memory. It will be appreciated that there will not typically be any physical distinction between the secure memory regions and non-secure memory regions of the memory at the time of fabrication, but that these regions will instead be defined by a secure operating system of the data processing apparatus when operating in the secure domain. Hence, any physical part of the memory device may be allocated as secure memory, and any physical part may be allocated as non-secure memory.

As described with reference to Figures 2 to 5, the processing system has a secure domain and a non-secure domain. In the secure domain, a secure kernel program 80 is provided and which executes in a secure mode. A monitor program 72 is provided which straddles the secure and non-secure domains and which executes at least partly in a monitor mode. In embodiments of the invention the monitor program executes partly in the monitor mode and partly in a secure mode. As shown in for example Figure 10, there are a plurality of secure modes including, inter alia, a supervisor mode SVC.

The monitor program 72 is responsible for managing all changes between the secure and non-secure domains in either direction. Some of its functions are

The apparatus has a processor core 10 which defines the modes and defines the privilege levels of the modes; i.e. the set of functions which any mode allows. Thus the processor core 10 is arranged in known manner to allow the secure modes and the monitor mode access to secure and non-secure memory and the secure modes access  
5 to all memory to which the monitor mode allows access and to allow a process operating in any privileged secure mode to switch directly to monitor mode and vice versa. The processor core 10 is preferably arranged to allow the following.

In one example of the apparatus, the memory is partitioned into secure  
10 memory and non-secure memory, and both secure and non-secure memory is accessible only in the monitor and secure modes. Preferably, the non-secure memory is accessible in monitor mode, a secure mode and a non-secure mode.

In another example of the apparatus, in the monitor mode and one or more of  
15 the secure modes, access to the non-secure memory is denied to the secure mode; and in non-secure mode access to the non-secure memory is denied to the secure and monitor modes. Thus secure memory is accessed only in monitor and secure modes and non-secure memory is accessed only by non-secure modes increasing security.

In examples of the apparatus, resetting or booting of the apparatus may be  
20 performed in the monitor mode which may be regarded as a mode which is more privileged than a secure mode. privileged mode. However, in many examples of the apparatus are arranged to provide resetting or booting in a secure mode which is possible because of the direct switching allowed between the secure mode and the  
25 monitor mode.

As described with reference to Figure 2, in the secure domain, and in a secure mode, a secure kernel 80 (or operating system) functions, and one or more secure application programs 82, 84 may be run under the secure kernel 80. The secure kernel  
30 and/or the secure application program or any other program code running in a secure mode is allowed access to both secure and non-secure memory.

runs in non-secure world, each time a secure function is called, it will be necessary to pass as a parameter the current thread ID to link the secure function to its calling non-secure application. The secure world can thus support multi-threads.

- Secure Interrupt defines an interrupt generated by a Secure peripheral.

5

### **Programmer's model**

#### **Carbon Core Overview**

The concept of the Carbon architecture, which is the term used herein for processors using the present techniques, consists in separating two worlds, one secure and one non-secure. The secure world must not leak any data to non-secure world.

In the proposed solution, the secure and non-secure states will share the same (existing) register bank. As a consequence, all current modes present in ARM cores (Abort, Undef, Irq, User, ...) will exist in each state.

The core will know it operates in secure or non-secure state thanks to a new state bit, the S (secure) bit, instantiated in a dedicated CP15 register.

Controlling which instruction or event is allowed to modify the S bit, i.e. to change from one state to the other, is a key feature of the security of the system. The current solution proposes to add a new mode, the Monitor mode, that will "supervise" switching between the two states. The Monitor mode, by writing to the appropriate CP15 register, would be the only one allowed to alter the S bit.

25

Finally, we propose to add some flexibility to the exception handling. All exceptions, apart from the reset, would be handled either in the state where they happened, or would be directed to the Monitor mode. This would be left configurable thanks to a dedicated CP15 register.

30

The details of this solution are discussed in the following paragraphs.

Any Secure privileged mode (i.e. privileged modes when S=1) would be able to switch to Monitor mode by simply writing the CPSR mode bits (MSR, MOVS, or equivalent instruction). However, this would be forbidden in any Non-secure mode or Secure user mode. If this ever happens, the instruction would be ignored or cause an exception.

There may be a need for a dedicated CPSR violation exception. This exception would be raised by any attempt to switch to Monitor mode by directly writing the CPSR from any Non-secure mode or Secure user mode.

All exceptions except Reset are in effect disabled when Monitor mode is active:

- all interrupts are masked;
- all memory exceptions are either ignored or cause a fatal exception.
- undefined/SWI/SMI are ignored or cause a fatal exception.

When entering Monitor mode, the interrupts are automatically disabled and the system monitor should be written such that none of the other types of exception can happen while the system monitor is running.

Monitor mode needs to have some private registers. This solution proposes that we only duplicate the minimal set of registers, i.e R13 (sp\_mon), R14 (lr\_mon) and SPSR (spsr\_mon).

In Monitor mode, the MMU will be disabled (flat address map) as well as the MPU or partition checker (the Monitor mode will always perform secure privileged external accesses). However, specially programmed MPU region attributes (cacheability, ...) would still be active. As an alternative the Monitor mode may use whatever mappings is used by the secure domain.

**New instruction**

flushed when passing from Secure to Non-secure state in order to avoid any leak of Secure data. This will need to be ensured by the Monitor kernel.

5 The possibility of implementing a hardware mechanism or a new instruction to directly flush the registers when switching from Secure to Non-secure state is also a possibility.

10 Another solution proposed involves duplicating all (or most of) the existing register bank, thus having two physically separated register banks between the Secure and Non-secure state. This solution has the main advantage of clearly separating the secure and non-secure data contained in the registers. It also allows fast context switching between the secure and non-secure states. However, the drawback is that passing parameters through registers becomes difficult, unless we create some dedicated instructions to allow the secure world access the non-secure registers

15

Figure 22 illustrates the available registers depending on the processor mode. Note that the processor state has no impact on this topic.

### Exceptions

20

#### Secure interrupts

#### Current Solution

It is currently proposed to keep the same interrupt pins as in the current cores, i.e. IRQ and FIQ. In association with the Exception Trap Mask register (defined later in the document), there should be sufficient flexibility for any system to implement and handle different kind of interrupts.

25

### VIC enhancement

30 We could enhance the VIC (Vectored Interrupt Controller) in the following way: the VIC may contain one Secure information bit associated to each vectored address. This bit would be programmable by the Monitor or Secure privileged modes only. It would indicate whether the considered interrupt should be treated as Secure, and thus should be handled on the Secure side.

<b>Non-Secure Interrupt</b>	<p>The VIC has no Vector associated to this interrupt in the Secure domain. It thus presents to the core the address contained in the Vector address register dedicated to all Non-secure interrupts occurring in Secure world. The core, still in Secure world, then branches to this address, where it should find an SMI instruction to switch to Non-secure world. Once in Non-secure world, it would be able to have access to the correct ISR.</p>	<p>No need to switch between worlds. The VIC directly presents to the core the Non-secure address associated to the interrupt line. The core simply has to branch at this address where it should find the associated Non-secure ISR.</p>
---------------------------------	--	---

### Exception handling configurability

In order to improve Carbon flexibility, a new register, the Exception Trap Mask, would be added in CP15. This register would contain the following bits:

- 5 - Bit 0: Undef exception (Non-secure state)
- Bit 1: SWI exception (Non-secure state)
- Bit 2: Prefetch abort exception (Non-secure state)
- Bit 3: Data abort exception (Non-secure state)
- Bit 4: IRQ exception (Non-secure state)
- 10 - Bit 5: FIQ exception (Non-secure state)
- Bit 6: SMI exception (both Non-secure/Secure states)
- Bit 16: Undef exception (Secure state)
- Bit 17: SWI exception (Secure state)
- Bit 18: Prefetch abort exception (Secure state)
- 15 - Bit 19: Data abort exception (Secure state)
- Bit 20: IRQ exception (Secure state)
- Bit 21: FIQ exception (Secure state)

0x08	SWI	Supervisor	SWI instruction executed when core is in Non-Secure state and Exception Trap Mask reg [Non-secure SWI]=0
0x0C	Prefetch Abort	Abort	Aborted instruction when core is in Non-Secure state and Exception Trap Mask reg [Non-secure PAbort]=0
0x10	Data Abort	Abort	Aborted data when core is in Non-Secure state and Exception Trap Mask reg [Non-secure DAbort]=0
0x14	Reserved		
0x18	IRQ	IRQ	IRQ pin asserted when core is in Non-Secure state and Exception Trap Mask reg [Non-secure IRQ]=0
0x1C	FIQ	FIQ	FIQ pin asserted when core is in Non-Secure state and Exception Trap Mask reg [Non-secure FIQ]=0

**In secure memory:**

Address	Exception	Mode	Automatically accessed when
0x00	Reset*	Supervisor	Reset pin asserted
0x04	Undef	Undef	Undefined instruction executed when core is in Secure state and Exception Trap Mask reg [Secure Undef]=0
0x08	SWI	Supervisor	SWI instruction executed when core is in Secure state and Exception Trap Mask reg [Secure SWI]=0
0x0C	Prefetch Abort	Abort	Aborted instruction when core is in Secure state and Exception Trap Mask reg [Secure PAbort]=0

0x10	Data Abort	Monitor	Aborted data when Core is in Secure state and Exception Trap Mask reg [Secure PAbort]=1 Core is in Non-secure state and Exception Trap Mask reg [Non-secure Pabort]=1
0x14	SMI	Monitor	
0x18	IRQ	Monitor	- IRQ pin asserted when core is in Secure state and Exception Trap Mask reg [Secure IRQ]=1 core is in Non-secure state and Exception Trap Mask reg [Non-secure IRQ]=1
0x1C	FIQ	Monitor	- FIQ pin asserted when core is in Secure state and Exception Trap Mask reg [Secure FIQ]=1 core is in Non-secure state and Exception Trap Mask reg [Non-secure FIQ]=1

In Monitor mode, the exceptions vectors may be duplicated, so that each exception will have two different associated vector:

- One for the exception arising in Non-secure state
- 5 - One for the exception arising in Secure state

This may be useful to reduce the exception latency, because the monitor kernel does not have any more the need to detect the originating state where the exception occurred.

- 10 Note that this feature may be limited to a few exceptions, the SMI being one of the most suitable candidates to improve the switching between the Secure and Non-secure states.

### Switching between worlds



1. Thread 1 is running in non-secure world (S bit = 0)

This thread needs to perform a secure function => SMI instruction.

2. The SMI instruction makes the core enter the Monitor mode through a non-secure SMI vector.

- 5 LR\_mon and SPSR\_mon are used to save the PC and CPSR of the non secure mode.

At this stage the S bit remains unchanged, although the system is now in a secure state.

The monitor kernel saves the non-secure context on the monitor.

It also pushes LR\_mon and SPSR\_mon.

- 10 The monitor kernel then changes the "S" bit by writing into the CP15 register.

In this embodiment the monitor kernel keeps track that a "secure thread 1" will be started in the secure world (e.g. by updating a Thread ID table).

Finally, it exits the monitor mode and switches to secure supervisor mode (MOVS instruction after having updated LR\_mon and SPSR\_mon?).

- 15 3. The secure kernel dispatches the application to the right secure memory location, then switches to user mode (e.g. using a MOVS).

4. The secure function is executed in secure user mode. Once finished, it calls an "exit" function by performing an appropriate SWI.

5. The SWI instruction makes the core enter the secure svc mode through a  
20 dedicated SWI vector, that in turn performs the "exit" function. This "exit" function ends with an "SMI" to switch back to monitor mode.

6. The SMI instruction makes the core enter the monitor mode through a dedicated secure SMI vector.

- 25 LR\_mon and SPSR\_mon are used to save the PC and CPSR of the Secure svc mode.

The S bit remains unchanged (i.e. Secure State).

The monitor kernel registers the fact that secure thread 1 is finished (removes the secure thread 1 ID from the thread ID table?).

- 30 It then changes the "S" bit by writing into the CP15 register, returning to non-secure state.

The monitor kernel restores the non-secure context from the monitor stack.

It also loads the LR\_mon and CPSR\_mon previously saved in step 2.

data values associated with one domain and stored in the said disabled register and to load into that register new data values associated with the other domain and then re-enable the FIQ mode registers.

5       The processor may be arranged so that when in the monitor mode all banked registers are disabled when the processor switches domains. Alternatively, the disabling of the registers may be selective in that some predetermined ones of the shared registers are disabled when switching domains and others may be disabled at the choice of the programmer.

10

The processor may be arranged so that when switching domains in the monitor mode, one or more of the shared registers are disabled, and one or more others of the shared registers have their data saved when existing one domain, and have new data loaded in the other domain. The new data may be null data.

15

Figure 24 schematically illustrates the concept of adding a secure processing option to a traditional ARM core. The diagram schematically shows how a processor that contains a secure processing option can be formed by adding a secure processing option to an existing core. If the system is to be backwards compatible with an existing legacy operating system, it is intuitive to think of the legacy system operating in the traditional non-secure part of the processor. However, as is shown schematically in the lower part of the Figure and is detailed further below, it is in fact in the secure portion of the system that a legacy system operates.

25

Figure 25 shows a processor having a secure and non-secure domain and illustrating reset and is similar to Figure 2. Figure 2 illustrates a processor that is adapted to run a security sensitive type of operation with a secure OS system controlling processing in the secure domain and a non-secure OS system controlling processing in the non-secure domain. The processor is however also backwards compatible with a traditional legacy operating system and thus, the processor may operate in a security insensitive way using a legacy operating system.

30

Booting in secure supervisor mode in all cases is also advantageous in security sensitive systems as it helps ensure the security of the system. In secure sensitive systems, the address provided at boot points to where the boot program is stored in secure supervisor mode and thus, enables the system to be configured as a secure system and to switch to monitor mode. Switching from secure supervisor mode to monitor mode is allowed in general and enables the secure system at an appropriate time to start processing in monitor mode to initialise monitor mode configuration.

Figure 26 illustrates at step 1 a non-secure thread NSA being executed by a non-secure operating system. At step 2 the non-secure thread NSA makes a call to the secure domain via the monitor mode running a monitor mode program at step 3. The monitor mode program changes the S-bit to switch domain and performs any necessary context saving and context restoring prior to moving to the secure operating system at step 5. The corresponding secure thread SA is then executed before it is subject to an interrupt irq at step 6. The interrupt handling hardware triggers a return to the monitor mode at step 7 where it is determined as to whether the interrupt will be handled by the secure operating system or the non-secure operating system. In this case, the interrupt is handled by the non-secure operating system starting at step 9. When this interrupt has been handled by the non-secure operating system, the non-secure thread NSA is resumed as the current task in the non-secure operating system prior to a normal thread switching operation at step 11. This thread switching may be the result of a timing event or the like. A different thread NSB is executed in the non-secure domain by the non-secure operating system at step 12 and this then makes a call to the secure domain via the monitor domain/program at step 14. The monitor program at step 7 has stored a flag, used some other mechanism, to indicate that the secure operating system was last suspended as a result of an interrupt rather than having been left because a secure thread had finished execution or due to a normal request to leave. Accordingly, since the secure operating system was suspended by an interrupt, the monitor program at step 15 re-enters the secure operating system using a software faked interrupt which specifies a return thread ID (e.g. an identifier of the thread to be started by the secure operating system as requested by the non-secure

Figure 29 schematically illustrates processing whereby a slave secure operating system may follow task switches performed by a master non-secure operating system. The master non-secure operating system may be a legacy operating system with no mechanisms for communicating and co-ordinating its activities to other operating systems and accordingly operate only as a master. As an initial entry point in Figure 29 the non-secure operating system is executing a non-secure thread NSA. This non-secure thread NSA makes a call to a secure thread which is to be executed by the secure operating system using a software interrupt, an SMI call. The SMI call goes to a monitor program executing in a monitor mode at step 2 whereupon the monitor program performs any necessary context saving and switching before passing the call onto the secure operating system at step 4. The secure operating system then starts the corresponding secure thread SA. This secure thread may return control via the monitor mode to the non-secure operating system, such as as a result of a timer event or the like. When the non-secure thread NSA again passes control to the secure operating system at step 9 it does so by reissuing the original software interrupt. The software interrupt includes the non-secure thread ID identifying NSA, the secure thread ID of the target secure thread to be activated, i.e. the thread ID identifying secure thread SA as well as other parameters.

When the call generated at step 9 is passed on by the monitor program and received at step 12 in the secure domain by the secure operating system, the non-secure thread ID can be examined to determine whether or not there has been a context switch by the non-secure operating system. The secure thread ID of the target thread may also be examined to see that the correct thread under the secure operating system is restarted or started as a new thread. In the example of Figure29, no thread switch is required in the secure domain by the secure operating system.

Figure 30 is similar to Figure 29 except that a switch in thread occurs at step 9 in the non-secure domain under control of the non-secure operating system. Accordingly, it is a different non-secure thread NSB which makes the software interrupt call across to the secure operating system at step 11. At step 14, the secure operating system recognises the different thread ID of the non-secure thread NSB and

monitor program in the monitor mode used to determine where the interrupt Int2 is to be handled. In this case the interrupt Int2 is to be handled by the non-secure operating system and accordingly control is passed to the non-secure operating system and the interrupt handler for Int2 started. When this interrupt handler for the interrupt Int2 has completed, the non-secure operating system has no information indicating that there is a pending interrupt Int1 for which servicing has been suspended in the secure domain. Accordingly, the non-secure operating system may perform some further processing, such as a task switch or the starting of a different non-secure thread NSB, whilst the original interrupt Int1 remains unserviced.

Figure 33 illustrates a technique whereby the problems associated with the operation of Figure 32 may be avoided. When the interrupt Int1 occurs, the monitor program passes this to the non-secure domain where a stub interrupt handler is started. This stub interrupt handler is relatively small and quickly returns processing to the secure domain via the monitor mode and triggers an interrupt handler for the interrupt Int1 within the secure domain. The interrupt Int1 is primarily processed within the secure domain and the starting of the stub interrupt handler in the non-secure domain can be regarded as a type of placeholder to indicate to the non-secure domain that the interrupt is pending in the secure domain.

The interrupt handler in the secure domain for the interrupt Int1 is again subject to a high priority Int2. This triggers execution of the interrupt handler for the interrupt Int2 in the non-secure domain as before. However, in this case, when that interrupt handler for Int2 has finished, the non-secure operating system has data indicating that the stub interrupt handler for interrupt Int1 is still outstanding and accordingly will resume this stub interrupt handler. This stub interrupt handler will appear as if it were suspended at the point at which it made its call back to the secure domain and accordingly this call will be re-executed and thus the switch made to the secure domain. Once back in the secure domain, the secure domain can itself re-start the interrupt handler for the interrupt Int1 at the point at which it was suspended. When the interrupt handler for the interrupt Int1 has completed within the secure domain, a call is made back to the non-secure domain to close down the stub interrupt

As illustrated in Figure 35, this processor configuration data is stored within the CP15 registers 34, and in one embodiment these registers are shared between the domains. Hence, when the monitor mode is switched between the secure domain and the non-secure domain, the processor configuration data currently in the CP15 registers 34 needs to be switched out of the CP15 registers into memory, and processor configuration data relating to the destination domain needs to be loaded into the CP15 registers 34.

Since the processor configuration data in the CP15 registers typically has an immediate effect on the access to memory within the system, then it is clear that these settings would become immediately effective as they are updated by the processor whilst operating in the monitor mode. However, this is undesirable since it is desirable for the monitor mode to have a static set of processor configuration data that control access to memory whilst in monitor mode.

Accordingly, as shown in Figure 35, in one embodiment of the present invention monitor mode specific processor configuration data 2000 is provided, which can be used to override the processor configuration data in the CP15 registers 34 whilst the processor is operating in the monitor mode. This is achieved in the embodiment illustrated in Figure 35 through the provision of a multiplexer 2010 which receives at its inputs both the processor configuration data stored in the CP15 registers and the monitor mode specific processor configuration data 2000. Furthermore, the multiplexer 2010 receives a control signal over path 2015 indicating whether the processor is currently operating in the monitor mode or not. If the processor is not operating in the monitor mode, then the processor configuration data in the CP15 registers 34 is output to the system, but in the event that the processor is operating in the monitor mode, the multiplexer 2010 instead outputs the monitor mode specific processor configuration data 2000 to ensure that a consistent set of processor configuration data is applied while the processor is operating in the monitor mode.

The monitor mode specific processor configuration data may also specify other data used to control access to parts of the memory. For example, the monitor mode specific processor configuration data may specify that the cache 38 is not to be used to  
5 access data whilst the processor is operating in the monitor mode.

In the embodiment described above, it has been assumed that all of the CP15 registers containing processor configuration data are shared between the domains. However, in an alternative embodiment, a number of the CP15 registers are “banked”,  
10 so that for example there are two registers for storing a particular item of processor configuration data, one register being accessible in the non-secure domain and containing the value of that item of processor configuration data for the non-secure domain, and the other register being accessible in the secure domain and containing the value of that item of processor configuration data for the secure domain.

15 One CP15 register that will not be banked is the one containing the “S” bit, but in principle any of the other CP15 registers may be banked if desired. In such embodiments, the switching of the processor configuration data by the monitor mode involves switching out of any shared CP15 registers into memory the processor  
20 configuration data currently in those shared registers, and loading into those shared CP15 registers the processor configuration data relating to the destination domain. For any banked registers, the processor configuration data need not be stored away to memory, and instead the switching will occur automatically as a result of changing the S bit value stored in the relevant shared CP15 register.

25 As mentioned earlier, the monitor mode processor configuration data will include a domain status bit which overrides that stored in the relevant CP15 register but has the same value as that used for the domain status bit used in the secure domain (i.e. an S bit value of 1 in the above described embodiments). When a number of the  
30 CP15 registers are banked, this means that at least part of the monitor mode specific processor configuration data 2000 in Figure 35 can be derived from the secure

portions of memory allocated for storing state information, one allocated for storing the state for the non-secure domain, and one allocated for storing the state for the secure domain.

5        Once the state pointer has been switched at step 2050, that state now pointed to by the state pointer is loaded into the relevant shared CP15 registers at step 2060, this including loading in the relevant processor configuration data for the destination domain. Thereafter, at step 2070, the monitor program is exited, as is the monitor mode, and the processor then switches to the required mode in the destination domain.

10

Figure 37 illustrates in more detail the operation of the memory management logic 30 of one embodiment of the present invention. The memory management logic consists of a Memory Management Unit (MMU) 200 and a Memory Protection Unit (MPU) 220. Any memory access request issued by the core 10 that specifies a virtual address will be passed over path 234 to the MMU 200, the MMU 200 being responsible for performing predetermined access control functions, more particularly for determining the physical address corresponding to that virtual address, and for resolving access permission rights and determining region attributes.

15

20        The memory system of the data processing apparatus consists of secure memory and non-secure memory, the secure memory being used to store secure data that is intended only to be accessible by the core 10, or one or more other master devices, when that core or other device is operating in a secure mode of operation, and is accordingly operating in the secure domain.

25

In the embodiment of the present invention illustrated in Figure 37, the policing of attempts to access secure data in secure memory by applications running on the core 10 in non-secure mode is performed by the partition checker 222 within the MPU 220, the MPU 220 being managed by the secure operating system, also referred to herein as the secure kernel.

30



In the event that there is no match found within the micro-TLB 206, then the memory access request is passed over path 242 to the main TLB 208 which contains a number of descriptors obtained from the page tables. As will be discussed in more detail later, descriptors from both the non-secure page table and the secure page table can co-exist within the main TLB 208, and each entry within the main TLB has a corresponding flag (referred to herein as a domain flag) which is settable to indicate whether the corresponding descriptor in that entry has been obtained from a secure page table or a non-secure page table. In any embodiments where all secure modes of operation specify physical addresses directly within their memory access requests, it will be appreciated that there will not be a need for such a flag within the main TLB, as the main TLB will only store non-secure descriptors.

Within the main TLB 208, a similar lookup process is performed to determine whether the relevant portion of the virtual address issued within the memory access request corresponds with any of the virtual address portions associated with descriptors in the main TLB 208 that are relevant to the particular mode of operation. Hence, if the core 10 is operating in non-secure mode, only those descriptors within the main TLB 208 which have been obtained from the non-secure page table will be checked, whereas if the core 10 is operating in secure mode, only the descriptors within the main TLB that have been obtained from the secure page table will be checked.

If there is a hit within the main TLB as a result of that checking process, then the access control information is extracted from the relevant descriptor and passed back over path 242. In particular, the virtual address portion and the corresponding physical address portion of the descriptor will be routed over path 242 to the micro-TLB 206, for storage in an entry of the micro-TLB, the access permission rights will be loaded into the access permission logic 202, and the region attributes will be loaded into the region attribute logic 204. The access permission logic 202 and region attribute logic 204 may be separate to the micro-TLB, or may be incorporated within the micro-TLB.

walk process needs to be performed, the translation table walk logic 210 will know in which domain the core 10 is executing, and accordingly which base address to use to access the relevant table. The virtual address is then used as an offset to the base address in order to access the appropriate entry within the appropriate page table in  
5 order to obtain the required descriptor.

Once the descriptor has been retrieved by the translation table walk logic 210, and placed within the main TLB 208, a hit will then be obtained within the main TLB, and the earlier described process will be invoked to retrieve the access control  
10 information, and store it within the micro-TLB 206, the access permission logic 202 and the region attribute logic 204. The memory access can then be actioned by the MMU 200.

As mentioned earlier, in preferred embodiments, the main TLB 208 can store  
15 descriptors from both the secure page table and the non-secure page table, but the memory access requests are only processed by the MMU 200 once the relevant information is stored within the micro-TLB 206. In preferred embodiments, the transfer of information between the main TLB 208 and the micro-TLB 206 is monitored by the partition checker 222 located within the MPU 220 to ensure that, in  
20 the event that the core 10 is operating in a non-secure mode, no access control information is transferred into the micro-TLB 206 from descriptors in the main TLB 208 if that would cause a physical address to be generated which is within secure memory.

25 The memory protection unit is managed by the secure operating system, which is able to set within registers of the CP15 34 partitioning information defining the partitions between the secure memory and the non-secure memory. The partition checker 222 is then able to reference that partitioning information in order to determine whether access control information is being transferred to the micro-TLB  
30 206 which would allow access by the core 10 in a non-secure mode to secure memory. More particularly, in preferred embodiments, when the core 10 is operating in a non-secure mode of operation, as indicated by the domain bit set by the monitor mode

operation the MMU 200 will be disabled, and the physical address will pass over path 236 into the MPU 220. In a secure mode of operation, the access permission logic 224 and the region attribute logic 226 will perform the necessary access permission and region attribute analysis based on the access permission rights and region attributes identified for the corresponding regions within the partitioning information registers within the CP15 34. If the secure memory location seeking to be accessed is within a part of secure memory only accessible in a certain mode of operation, for example secure privileged mode, then an access attempt by the core in a different mode of operation, for example a secure user mode, will cause the access permission logic 224 to generate an abort over path 230 to the core in the same way that the access permission logic 202 of the MMU would have produced an abort in such circumstances. Similarly, the region attribute logic 226 will generate cacheable and bufferable signals in the same way that the region attribute logic 204 of the MMU would have generated such signals for memory access requests specified with virtual addresses. Assuming the access is allowed, the access request will then proceed over path 240 onto the system bus 40, from where it is routed to the appropriate memory unit.

For a non-secure access where the access request specifies a physical address, the access request will be routed via path 236 into the partition checker 222, which will perform partition checking with reference to the partitioning information in the CP15 registers 34 in order to determine whether the physical address specifies a location within secure memory, in which event the abort signal will again be generated over path 230.

25

The above described processing of the memory management logic will now be described in more detail with reference to the flow diagrams of Figures 39 and 40. Figure 39 illustrates the situation in which the program running on the core 10 generates a virtual address, as indicated by step 300. The relevant domain bit within the CP15 domain status register 34 as set by the monitor mode will indicate whether the core is currently running in a secure domain or the non-secure domain. In the event that the core is running in the secure domain, the process branches to step 302,

descriptor is present. If it is, then the process branches directly to step 336, where the access permission rights are checked by the access permission logic 202. It is important to note at this point that if the relevant physical address portion is within the micro-TLB, it is assumed that there is no security violation, since the partition checker 222 effectively polices the information prior to it being stored within the micro-TLB, such that if the information is within the micro-TLB, it is assumed to be the appropriate non-secure information. Once the access permission has been checked at step 336, the process proceeds to step 338, where it is determined whether there is any violation, in which event an access permission fault abort is issued at step 316. Otherwise, the process proceeds to step 318 where the remainder of the memory access is performed, as discussed earlier.

In the event that at step 320 no hit was located in the micro-TLB, the process proceeds to step 322, where a lookup process is performed in the main TLB 208 to determine whether the relevant non-secure descriptor is present. If not, a page table walk process is performed at step 324 by the translation table walk logic 210 in order to retrieve into the main TLB 208 the necessary non-secure descriptor from the non-secure page table. The process then proceeds to step 326, or proceeds directly to step 326 from step 322 in the event that a hit within the main TLB 208 occurred at step 322. At step 326, it is determined that the main TLB now contains the valid tagged non-secure descriptor for the virtual address in question, and then at step 328 the partition checker 222 checks that the physical address that would be generated from the virtual address of the memory access request (given the physical address portion within the descriptor) will point to a location in non-secure memory. If not, i.e. if the physical address points to a location in secure memory, then at step 330 it is determined that there is a security violation, and the process proceeds to step 332 where a secure/non-secure fault abort is issued by the partition checker 222.

If however the partition checker logic 222 determines that there is no security violation, the process proceeds to step 334, where the micro-TLB is loaded with the sub-section of the relevant descriptor that contains the physical address portion,

accordingly physical addresses are not generated directly in any of the modes of operation. In this scenario, it will be appreciated that a separate MPU 220 is not required, and instead the partition checker 222 can be incorporated within the MMU 200. This change aside, the processing proceeds in exactly the same manner as discussed earlier with reference to Figures 37 and 39.

It will be appreciated that various other options are also possible. For example, assuming memory access requests may be issued by both secure and non-secure modes specifying virtual addresses, two MMUs could be provided, one for secure access requests and one for non-secure access requests, i.e. MPU 220 in Figure 37 could be replaced by a complete MMU. In such cases, the use of flags with the main TLB of each MMU to define whether descriptors are secure or non-secure would not be needed, as one MMU would store non-secure descriptors in its main TLB, and the other MMU would store secure descriptors in its main TLB. Of course, the partition checker would still be required to check whether an access to secure memory is being attempted whilst the core is in the non-secure domain.

If, alternatively, all memory access requests directly specified physical addresses, an alternative implementation might be to use two MPUs, one for secure access requests and one for non-secure access requests. The MPU used for non-secure access requests would have its access requests policed by a partition checker to ensure accesses to secure memory are not allowed in non-secure modes.

As a further feature which may be provided with either the Figure 37 or the Figure 38 arrangement, the partition checker 222 could be arranged to perform some partition checking in order to police the activities of the translation table walk logic 210. In particular, if the core is currently operating in the non-secure domain, then the partition checker 222 could be arranged to check, whenever the translation table walk logic 210 is seeking to access a page table, that it is accessing the non-secure page table rather than the secure page table. If a violation is detected, an abort signal would preferably be generated. Since the translation table walk logic 210 typically performs the page table lookup by combining a page table base address with certain bits of the

whereas when in secure mode, the secure page table will be referenced by the translation table walk logic 210. Figure 42 illustrates these two page tables. As shown in Figure 42, the non-secure memory 390, which may for example be within external memory 56 of Figure 1, includes within it a non-secure page table 395 specified in a CP15 register 34 by reference to a base address 397. Similarly, within secure memory 400, which again may be within the external memory 56 of Figure 1, a corresponding secure page table 405 is provided which is specified within a duplicate CP15 register 34 by a secure page table base address 407. Each descriptor within the non-secure page table 395 will point to a corresponding non-secure page in non-secure memory 390, whereas each descriptor within the secure page table 405 will define a corresponding secure page in the secure memory 400. In addition, as will be described in more detail later, it is possible for certain areas of memory to be shared memory regions 410, which are accessible by both non-secure modes and secure modes.

Figure 43 illustrates in more detail the lookup process performed within the main TLB 208 in accordance with preferred embodiments. As mentioned earlier, the main TLB 208 includes a domain flag 425 which identifies whether the corresponding descriptor 435 is from the secure page table or the non-secure page table. This ensures that when a lookup process is performed, only the descriptors relevant to the particular domain in which the core 10 is operating will be checked. Figure 43 illustrates an example where the core is running in the secure domain, also referred to as the secure world. As can be seen from Figure 43, when a main TLB 208 lookup is performed, this will result in the descriptors 440 being ignored, and only the descriptors 445 being identified as candidates for the lookup process.

In accordance with preferred embodiments, an additional process ID flag 430, also referred to herein as the ASID flag, is provided to identify descriptors from process specific page tables. Accordingly, processes P1, P2 and P3 may each have corresponding page tables provided within the memory, and further may have different page tables for non-secure operation and secure operation. Further, it will be appreciated that the processes P1, P2, P3 in the secure domain may be entirely

In preferred embodiments, the choice between this configuration of the main TLB, and the earlier described configuration with separate secure and non-secure descriptors, can be set by a particular bit provided within the CP15 control registers. In preferred embodiments, this bit would only be set by the secure kernel.

5

In embodiments where the secure application were directly allowed to use a non-secure virtual address, it would be possible to make a non-secure stack pointer available from the secure domain. This can be done by copying a non-secure register value identifying the non-secure stack pointer into a dedicated register within the CP15 registers 34. This will then enable the non-secure application to pass parameters via the stack according to a scheme understood by the secure application.

As described earlier, the memory may be partitioned into non-secure and secure parts, and this partitioning is controlled by the secure kernel using the CP15 registers 34 dedicated to the partition checker 222. The basic partitioning approach is based on region access permissions as definable in typical MPU devices. Accordingly, the memory is divided into regions, and each region is preferably defined with its base address, size, memory attributes and access permissions. Further, when overlapping regions are programmed, the attributes of the upper region take highest priority. Additionally, in accordance with preferred embodiments of the present invention, a new region attribute is provided to define whether that corresponding region is in secure memory or in non-secure memory. This new region attribute is used by the secure kernel to define the part of the memory that is to be protected as secure memory.

25

At the boot stage, a first partition is performed as illustrated in Figure 44. This initial partition will determine the amount of memory 460 allocated to the non-secure world, non-secure operating system and non-secure applications. This amount corresponds to the non-secure region defined in the partition. This information will then be used by the non-secure operating system for its memory management. The rest of the memory 462, 464, which is defined as secure, is unknown by the non-secure operating system. In order to protect integrity in the non-secure world, the non-

30

However, one problem that could occur would be for an application in the non-secure domain to be able to use the cache operations register to invalidate, clean, or flush the cache. It needs to be ensured that such operations could not affect the security of the system. For example, if the non-secure operating system were to invalidate the cache 38 without cleaning it, any secure dirty data must be written to the external memory before being replaced. Preferably, secure data is tagged in the cache, and accordingly can be dealt with differently if desired.

10 In preferred embodiments, if an "invalidate line by address" operation is executed by a non-secure program, the physical address is checked by the partition checker 222, and if the cache line is a secure cache line, the operation becomes a "clean and invalidate" operation, thereby ensuring that the security of the system is maintained. Further, in preferred embodiments, all "invalidate line by index" operations that are executed by a non-secure program become "clean and invalidate by index" operations. Similarly, all "invalidate all" operations executed by a non-secure program become "clean and invalidate all" operations.

Furthermore, with reference to Figure 1, any access to the TCM 36 by the DMA 32 is controlled by the micro-TLB 206. Hence, when the DMA 32 performs a lookup in the TLB to translate its virtual address into a physical one, the earlier described flags that were added in the main TLB allow the required security checking to be performed, just as if the access request had been issued by the core 10. Further, as will be discussed later, a replica partition checker is coupled to the external bus 70, preferably being located within the arbiter/decoder block 54, such that if the DMA 32 directly accesses the memory coupled to the external bus 70 via the external bus interface 42, the replica partition checker connected to that external bus checks the validity of the access. Furthermore, in certain preferred embodiments, it would be possible to add a bit to the CP15 registers 34 to define whether the DMA controller 32 can be used in the non-secure domain, this bit only being allowed to be set by the secure kernel when operating in a privileged mode.



By default, it is envisaged that the TCM would be used only by non-secure operating systems, as in this scenario the non-secure operating system would not need to be changed.

5 As mentioned earlier, in addition to the provision of the partition checker 222 within the MPU 220, preferred embodiments of the present invention also provide an analogous partition checking block coupled to the external bus 70, this additional partition checker being used to police accesses to memory by other master devices, for example the digital signal processor (DSP) 50, the DMA controller 52 coupled  
10 directly to the external bus, the DMA controller 32 connectable to the external bus via the external bus interface 42, etc. As mentioned earlier, the entire memory system can consist of several memory units, and a variety of these may exist on the external bus 70, for example the external memory 56, boot ROM 44, or indeed buffers or registers 48, 62, 66 within peripheral devices such as the screen driver 46, I/O interface 60, key  
15 storage unit 64, etc. Furthermore, different parts of the memory system may need to be defined as secure memory, for example it may be desired that the key buffer 66 within the key storage unit 64 should be treated as secure memory. If an access to such secure memory were to be attempted by a device coupled to the external bus, then it is clear that the earlier described memory management logic 30 provided  
20 within the chip containing the core 10 would not be able to police such accesses.

Figure 47 illustrates how the additional partition checker 492 coupled to the external bus, also referred to herein as the device bus, is used. The external bus would typically be arranged such that whenever memory access requests were issued onto  
25 that external bus by devices, such as devices 470, 472, those memory access requests would also include certain signals on the external bus defining the mode of operation, for example privileged, user, etc. In accordance with preferred embodiments of the present invention the memory access request also involves issuance of a domain signal onto the external bus to identify whether the device is operating in secure mode  
30 or non-secure mode. This domain signal is preferably issued at the hardware level, and in preferred embodiments a device capable of operating in secure or non-secure domains will include a predetermined pin for outputting the domain signal onto path

the various memory devices, it may be possible to avoid the need for a partition checker 492 to be provided separately on the external bus.

5 In the embodiments described with reference to Figures 37 and 38, a single MMU, along with a single set of page tables, was used to perform virtual to physical address translation. With such an approach, the physical address space would typically be segmented between non-secure memory and secure memory in a simplistic manner such as illustrated in Figure 49. Here a physical address space 2100 includes an address space starting at address zero and extending to address Y for one of the memory units  
10 within the memory system, for example the external memory 56. For each memory unit, the addressable memory would typically be sectioned into two parts, a first part 2110 being allocated as non-secure memory and a second part 2120 being allocated as secure memory.

15 With such an approach, it will be appreciated that there are certain physical addresses which are not accessible to particular domain(s), and these gaps would be apparent to the operating system used in those domain(s). Whilst the operating system used in the secure domain will have knowledge of the non-secure domain, and hence will not be concerned by this, the operating system in the non-secure domain should  
20 ideally not need to have any knowledge of the presence of the secure domain, but instead should operate as though the secure domain were not there.

As a further issue, it will be appreciated that a non-secure operating system will see its address space for the external memory as starting at address zero and extending to  
25 address X, and the non-secure operating system need know nothing about the secure kernel and in particular the presence of the secure memory extending from address X+1 up to address Y. In contrast, the secure kernel will not see its address space beginning at address zero, which is not what an operating system would typically expect.

30 One embodiment which alleviates the above concerns by allowing the secure memory regions to be completely hidden from the non-secure operating system's view of its physical address space, and by enabling both the secure kernel in the secure

constructed in a similar manner to the MMU 200 shown in Figure 37, but for the sake of ease of illustration certain detail has been omitted in Figure 50A.

5 The first MMU 2150 includes a micro-TLB 2155, a main TLB 2160 and translation table walk logic 2165, while similarly the second MMU 2170 includes a micro-TLB 2175, a main TLB 2180 and translation table walk logic 2185. The first MMU may be controlled by the non-secure operating system when the processor is operating in the non-secure domain, or by the secure kernel when the processor is operating in the secure domain. However, in preferred embodiments, the second MMU  
10 is only controllable by the secure kernel, or by the monitor program.

When the processor core 10 issues a memory access request, it will issue a virtual address over path 2153 to the micro-TLB 2155. The micro-TLB 2155 will store for a number of virtual address portions corresponding intermediate address portions  
15 retrieved from descriptors stored within the main TLB 2160, the descriptors in the main TLB 2160 having been retrieved from page tables in a first set of page tables associated with the first MMU 2150. If a hit is detected within the micro-TLB 2155, then the micro-TLB 2155 can issue over path 2157 an intermediate address corresponding to the virtual address received over path 2153. If there is no hit within the micro-TLB 2155,  
20 then the main TLB 2160 will be referenced to see if a hit is detected within the main TLB, and if so the relevant virtual address portion and corresponding intermediate address portion will be retrieved into the micro-TLB 2155, whereafter the intermediate address can be issued over path 2157.

25 If there is no hit within the micro-TLB 2155 and the main TLB 2160, then the translation table walk logic 2165 is used to issue a request for the required descriptor from a predetermined page table in a first set of page tables accessible by the first MMU 2150. Typically, there may be page tables associated with individual processes for both secure domain or non-secure domain, and the intermediate base addresses for those page  
30 tables will be accessible by the translation table walk logic 2165, for example from appropriate registers within the CP15 registers 34. Accordingly, the translation table

- 3) Miss in main TLB of MMU 1

Page Table 1 Base Address = 8000 IA [PA = 10000]

- 4) Translation Table Walk logic in MMU 1 performs page table lookup

- issues IA = 8003

- 5        5) Miss in micro-TLB of MMU 2

- 6) Miss in main TLB of MMU 2

Page Table 2 Base Address = 12000 PA

- 7) Translation Table Walk Logic in MMU 2 performs page table lookup

- issues PA = 12008

- 10      "8000 IA = 10000 PA" returned as page table data

- 8) - stored in main TLB of MMU 2

- 9) - stored in micro-TLB of MMU 2

- 10) Micro-TLB in MMU 2 now has hit

- issues PA = 10003

- 15      "3000 VA = 5000 IA" returned as page table data

- 11) - stored in main TLB of MMU 1

- 12) - stored in micro-TLB of MMU 1

- 13) Micro-TLB in MMU 1 now has hit

issues IA = 5000 to perform data access

- 20        14) miss in micro-TLB of MMU 2

- 15) miss in main TLB of MMU 2

- 16) Translation Table Walk Logic in MMU 2 performs page table lookup

- issues PA = 12005

"5000 IA = 7000 PA" returned as page table data

- 25        17) - stored in main TLB of MMU 2

- 18) - stored in micro-TLB of MMU 2

- 19) Micro-TLB in MMU 2 now has hit

- issues PA = 7000 to perform data access

- 20) Data at physical address 7000 returned to core

30

NEXT TIME CORE ISSUES A MEMORY ACCESS REQUEST (say VA 3001..)

Additionally, the use of such an approach considerably simplifies the process of swapping regions of memory between non-secure memory and secure memory. This is illustrated schematically with reference to Figure 52. As can be seen in Figure 52, a region of memory 2300, which may for example be a single page of memory, may exist within the non-secure memory region 2220, and similarly a memory region 2310 may exist within the secure memory region 2210. However, these two memory regions 2300 and 2310 can readily be swapped merely by changing the relevant descriptors within the second set of page tables, such that the region 2300 now becomes a secure region mapped to region 2305 in the intermediate address space of the secure domain, whilst region 2310 now becomes a non-secure region mapped to the region 2315 in the intermediate address space of the non-secure domain. This can occur entirely transparently to the operating systems in both the secure domain the non-secure domain, since their view of the physical address space is actually the intermediate address space of the secure domain or non-secure domain, respectively. Hence, this approach avoids any redefinition of the physical address space within each operating system.

An alternative embodiment of the present invention where two MMUs are also used, but in a different arrangement to that of Figure 50A, will now be described with reference to Figure 50B. As can be seen from a comparison of Figure 50B with Figure 50A, the arrangement is almost identical, but in this embodiment the first MMU 2150 is arranged to perform virtual address to physical address translation and the second MMU is arranged to perform intermediate address to physical address translation. Hence, instead of the path 2157 from the micro-TLB 2155 in the first MMU 2150 to the micro-TLB 2175 in the second MMU 2170 used in the Figure 50A embodiment, the micro-TLB 2155 in the first MMU is instead arranged to output a physical address directly over path 2192, as shown in Figure 50B. The operation of the embodiment illustrated in Figure 50B will now be illustrated by way of the specific example as set out below, which details the processing of the same core memory access request as illustrated earlier for the Figure 50A embodiment:

- 1) Core issues VA = 3000 [ IA = 5000, PA = 7000 ]
- 2) Miss in micro-TLB and main TLB of MMU 1

3) Data at PA 7001 is returned to the core.

As can be seen from a comparison of the above example with that provided for figure 50A, the main differences here are in step 7 where MMU1 does not store the first table descriptor directly, and in step 12b (12a and 12b can happen at the same time) where MMU1 also receives the IA->PA translation and does the combination and stores the combined descriptor in its TLBs.

Hence, it can be seen that whilst this alternative embodiment still uses the two sets of page tables to convert virtual addresses to physical addresses, the fact that the micro-TLB 2155 and main TLB 2160 store the direct virtual address to physical address translation avoids the need for lookups to be performed in both MMUs when a hit occurs in either the micro-TLB 2155 or the main TLB 2160. In such cases the first MMU can directly handle requests from the core without reference to the second MMU.

It will be appreciated that the second MMU 2170 could be arranged not to include the micro-TLB 2175 and the main TLB 2180, in which case the page table walk logic 2185 would be used for every request that needed handling by the second MMU. This would save on complexity and cost for the second MMU, and might be acceptable assuming the second MMU was only needed relatively infrequently. Since the first MMU will need to be used for every request, it will typically be expedient to include the micro-TLB 2155 and main TLB 2160 in the first MMU 2150 to improve speed of operation of the first MMU.

It should be noted that pages in the page tables may vary in size, and it is hence possible that the descriptors for the two halves of the translation relate to different sized pages. Typically, the MMU1 pages will be smaller than the MMU2 pages but this is not necessarily the case. For example:

Table 1 maps 4Kb at 0x40003000 onto 0x00081000  
Table 2 maps 1Mb at 0x00000000 onto 0x02000000

is a miss in the micro-TLB 2410, the main TLB 2420 is referenced and if the relevant descriptor is contained within the main TLB the associated virtual address portion and corresponding physical address portion are retrieved into the micro-TLB 2410, whereafter the physical address can be issued over path 2440. However, if the main  
5 TLB also produces a miss, then an exception is generated over path 2422 to the core. The process performed within the core from receipt of such an exception will now be described further with reference to Figure 54.

As shown in Figure 54, if a TLB miss exception is detected by the core at step  
10 2500, then the core enters the monitor mode at a predetermined vector for that exception at step 2510. This will then cause page table merging code to be run to perform the remainder of the steps illustrated in Figure 54.

More particularly, at step 2520, the virtual address that was issued over path  
15 2430, and that gave rise to the miss in both the micro-TLB 2410 and the main TLB 2420 (hereafter referred to as the faulting virtual address) is retrieved, whereafter at step 2530 the intermediate address for the required first descriptor is determined dependent on the intermediate base address for the appropriate table within the first set of tables. Once that intermediate address has been determined (typically by some predetermined  
20 combination of the virtual address with the intermediate base address), then the relevant table within the second set of tables is referenced in order to obtain the corresponding physical address for that first descriptor. Thereafter at step 2550 the first descriptor can be fetched from memory in order to enable the intermediate address for the faulting virtual address to be determined.

25

Then, at step 2560, the second table is again referenced to find a second descriptor giving the physical address for the intermediate address of the faulting virtual address. Thereafter at step 2570, the second descriptor is fetched to obtain the physical  
30 address for the faulting virtual address.

Once the above information has been obtained, then the program merges the first and second descriptors to generate a new descriptor giving the required virtual

performed in the micro-TLB 206 first, and accordingly access permissions, especially secure and non-secure permissions, would have been checked. Accordingly, in such embodiments, secure data cannot be stored in the cache 38 by non-secure applications. Access to the cache 38 is under the control of the partition checking performed by the partition checker 222, and accordingly no access to secure data can be performed in non-secure mode.

However, in an alternative embodiment of the present invention, a partition checker 222 is not provided for monitoring accesses made over the system bus 40, and instead the data processing apparatus merely has a single partition checker coupled to the external bus 70 for monitoring accesses to memory units connected to that external bus. In such embodiments, this then means that the processor core 10 can access any memory units coupled directly to the system bus 40, for example the TCM 36 and the cache 38, without those accesses being policed by the external partition checker, and accordingly some mechanism is required to ensure that the processor core 10 does not access secure data within the cache 38 or the TCM 36 whilst operating in a non-secure mode.

Figure 55 illustrates a data processing apparatus in accordance with one embodiment of the present invention, where a mechanism is provided to enable the cache 38 and/or the TCM 36 to control accesses made to them without the need for any partition checking logic to be provided in association with the MMU 200. As shown in Figure 55, the core 10 is coupled via an MMU 200 to the system bus 40, the cache 38 and TCM 36 also being coupled to the system bus 40. The core 10, cache 38 and TCM 36 are coupled via the external bus interface 42 to the external bus 70, which as illustrated in Figure 55 consists of an address bus 2620, a control bus 2630 and a data bus 2640.

The core 10, MMU 200, cache 38, TCM 36 and external bus interface 42 can be viewed as constituting a single device connected onto the external bus 70, also referred to as a device bus, and other devices may also be coupled to that device bus, for example the secure peripheral device 470 or the non-secure peripheral device 472. Also



the cache 38 will then perform a look-up process to determine whether the data item specified by that address is stored within the cache. Whenever a miss occurs within the cache, i.e. it is determined that the data item subject to the access request is not stored within the cache, a linefill procedure will be initiated by the cache in order to retrieve  
5 from the external memory 56 a line of data that includes the data item the subject of the memory access request. In particular, the cache will output via the EBI 42 a linefill request onto the control bus 2630 of the device bus 70, with a start address being output on the address bus 2620. In addition, an HPROT signal will be output over path 2632 onto the control bus 2630, which will include a domain signal specifying the mode of  
10 operation of the core at the time the memory access request was issued. Hence, the linefill process can be viewed as the propagation of the original memory access request onto the external bus by the cache 38.

This HPROT signal will be received by the partition checker 2656, and  
15 accordingly will identify to the partition checker whether the device requesting the specified data from the external memory 56 (in this case the device incorporating the core 10 and the cache 38) was operating in the secure domain or the non-secure domain at the time the memory access request was issued. The partition checker 2656 will also have access to the partitioning information identifying which regions of memory are  
20 secure or non-secure, and accordingly can determine whether the device is allowed to have access to the data it is requesting. Hence, the partition checker can be arranged to only allow a device to have access to a secure part of the memory if the domain signal within the HPROT signal (also referred to herein as an S bit) is asserted to identify that access to this data was requested by the device whilst operating in a secure mode of  
25 operation.

If the partition checker determines that the core 10 is not allowed to have access to the data requested, for example because the HPROT signal has identified that the core was operating in a non-secure mode of operation, but the linefill request is seeking to  
30 retrieve data from the external memory that is within a secure region of memory; then the partition checker 2656 will issue an abort signal onto the control bus 2630, which will be passed back over path 2636 to the EBI 42, and from there back to the cache 38,

The TCM 36 can be set up in a variety of ways. In one embodiment, it can be set up to act like a cache, and in that embodiment will be arranged to include a plurality of lines 2610, each of which has a flag 2612 associated therewith in the same way as the cache 38. Accesses to the TCM 36 are then managed in exactly the same way as described earlier with reference to the cache 38, with any TCM miss resulting in a linefill process being performed, as a result of which data will be retrieved into a particular line 2610, and the partition checker 2656 will generate the required S tag value for storing in the flag 2612 associated with that line 2610.

In an alternative embodiment, the TCM 36 may be set up as an extension of the external memory 56 and used to store data used frequently by the processor, since access to the TCM via the system bus is significantly faster than access to external memory. In such embodiments, the TCM 36 would not use the flags 2612, and instead a different mechanism would be used to control access to the TCM. In particular, as described earlier, in such embodiments, a control flag may be provided which is settable by the processor when operating in a privileged secure mode to indicate whether the tightly coupled memory is controllable by the processor only when executing in a privileged secure mode or is controllable by the processor when executing in the at least one non-secure mode. The control flag is set by the secure operating system, and in effect defines whether the TCM is controlled by the privileged secure mode or by non-secure modes. Hence, one configuration that can be defined is that the TCM is only controlled when the processor is operating in a privileged secure mode of operation. In such embodiments, any non-secure access attempted to the TCM control registers will cause an undefined instruction exception to be entered.

In an alternative configuration, the TCM can be controlled by the processor when operating in a non-secure mode of operation. In such embodiments, the TCM is only used by the non-secure applications. No secure data can be stored to or loaded from the TCM. Hence, when a secure access is performed, no look-up is performed within the TCM to see if the address matched the TCM address range.

If at step 2745 it was determined that the data item being requested is cacheable, then a cache look-up is performed at step 2750 within the cache, and if a hit is detected, the cache then determines whether there is a secure line tag violation at step 2755. Hence, at this stage, the cache will review the value of the flag 2602 associated with the  
5 cache line containing the data item, and will compare the value of that flag with the mode of operation of the core 10 to determine whether the core is entitled to access the data item requested. If a secure line tag violation is detected, then the process proceeds to step 2760, where a secure violation fault abort signal is generated by the cache 38 and issued over path 2670 to the core 10. However, assuming there is no secure line tag  
10 violation detected at step 2755, then the data access is performed at step 2785.

If when the cache look-up is performed at step 2750 a cache miss occurs, then a cache linefill is initiated at step 2765. At step 2770, the partition checker 2656 then detects whether there is a secure partition violation, and if so issues an abort signal at  
15 step 2775. However, assuming there is no secure partition violation detected, then the cache linefill proceeds at step 2780, resulting in the data access completing at step 2785.

As illustrated in Figure 57, steps 2705, 2710, 2715, 2720, 2725, 2730 and 2735 are performed within the MMU, steps 2745, 2750, 2755, 2765, 2780 and 2790 are  
20 performed by the cache, and steps 2770 and steps 2795 are performed by the partition checker.

Figure 58 is a flow diagram showing the analogous process performed in the event that a secure program executing on the core generates a virtual address (step  
25 2800). By comparison of Figure 58 with Figure 57, it will be appreciated that steps 2805 through 2835 performed within the MMU are analogous to the steps 2705 through 2735 described earlier with reference to Figure 57. The only difference is at step 2810, where the look-up performed within the main TLB is performed in relation to any secure descriptors stored within the main TLB, as a result of which at step 2820 the main TLB  
30 contains valid tagged secure descriptors.

state (via the JTAG serial interface) as well as the state of the memory system. This state is invasive to program execution, as it is possible to modify current mode, change register contents, etc. Once Debug is terminated, the core exits from the Debug State by scanning in the Restart instruction through the Debug TAP (test  
5 access port). Then the program resumes execution.

In monitor debug mode, a breakpoint or watchpoint causes the core to enter abort mode, taking prefetch or Data Abort vectors respectively. In this case, the core is still in a functional mode and is not stopped as it is in Halt debug mode. The abort  
10 handler communicates with a debugger application to access processor and coprocessor state or dump memory. A debug monitor program interfaces between the debug hardware and the software debugger. If bit 11 of the debug status and control register DSCR is set (see later), interrupts (FIQ and IRQ) can be inhibited. In monitor  
15 debug mode, vector catching is disabled on Data Aborts and Prefetch Aborts to avoid the processor being forced into an unrecoverable state as a result of the aborts that are generated for the monitor debug mode. It should be noted that monitor debug mode is a type of debug mode and is not related to monitor mode of the processor which is the mode that supervises switching between secure world and non-secure world.

20 Debug can provide a snapshot of the state of a processor at a certain moment. It does this by noting the values in the various registers at the moment that a debug initiation request is received. These values are recorded on a scan chain (541, 544 of Figure 67) and they are then serially output using a JTAG controller (18 or Figure 1).

25 An alternative way of monitoring the core is by trace. Trace is not intrusive and records subsequent states as the core continues to operate. Trace runs on an embedded trace macrocell (ETM) 22, 26 of Figure 1. The ETM has a trace port through which the trace information is exported, this is then analysed by an external trace port analyser.

30

The processor of embodiments of the present technique operates in two separate domains, in the embodiments described these domains comprise secure and

flushed in some embodiments when going from secure world to non-secure world to avoid any leak of data.

PC sample register: The Debug TAP can access the PC through scan chain 7.

- 5 When debugging in secure world, that value may be masked depending on the debug granularity chosen in secure world. It is important that non-secure world, or non-secure world plus secure user applications cannot get any value of the PC while the core is running in the secure world.

- 10 TLB entries: Using CP15 it is possible to read micro TLB entries and read and write main TLB entries. We can also control main TLB and micro TLB loading and matching. This kind of operation must be strictly controlled, particularly if secure thread-aware debug requires assistance of the MMU/MPU.

- 15 Performance Monitor Control register: The performance control register gives information on the cache misses, micro TLB misses, external memory requests, branch instruction executed, etc. Non-secure world should not have access to this data, even in Debug State. The counters should be operable in secure world even if debug is disabled in secure world.

- 20 Debugging in cache system: Debugging must be non-intrusive in a cached system. It is important is to keep coherency between cache and external memory. The Cache can be invalidated using CP15, or the cache can be forced to be write-through in all regions. In any case, allowing the modification of cache behaviour in  
25 debug can be a security weakness and should be controlled.

- Endianness: Non-secure world or secure user applications that can access to debug should not be allowed to change endianness. Changing the endianness could cause the secure kernel to malfunction. Endianness access is prohibited in debug,  
30 according to the granularity.

These control bits are stored in a register in the secure domain and access to this register is limited to three possibilities. Software access is provided via ARM coprocessor MRC/MCR instructions and these are only allowed from the secure supervisor mode. Alternatively, software access can be provided from any other mode with the use of an authentication code. A further alternative relates more to hardware access and involves the instructions being written via an input port on the JTAG. In addition to being used to input control values relating to the availability of monitoring functions, this input port can be used to input control values relating to other functions of the processor.

Further details relating to the scan chain and JTAG are given below.

#### **Register logic cell**

Every integrated circuit (IC) consists of two kind of logic:

- Combinatory logic cells; like AND, OR, INV gates. Such gates or combination of such gates is used to calculate Boolean expressions according to one or several input signals.
- Register logic cells; like LATCH, FLIP-FLOP. Such cells are used to memorize any signal value. Figure 62 shows a positive-edge triggered FLIP-FLOP view:

When positive-edge event occurs on the clock signal (CK), the output (Q) received the value of the input (D); otherwise the output (Q) keeps its value in memory.

#### **Scan chain cell**

For test or debug purpose, it is required to bypass functional access of register logic cells and to have access directly to the contents of the register logic cells. Thus register cells are integrated in a scan chain cell as shown in Figure 63.

mode, some scan chain cells (chosen by designer), may be “removed” from the scan chain structure. In order to keep the same number of scan-chain cell, the JTAG Selective Disable Scan Chain Cell use a bypass register. Note that Scan Out (SO) and scan chain cell output (Q) are now different.

5

Figure 67 schematically shows the processor including parts of the JTAG. In normal operation instruction memory 550 communicates with the core and can under certain circumstances also communicate with register CP14 and reset the control values. This is generally only allowable from secure supervisor mode.

10

When debug is initiated instructions are input via debug TAP 580 and it is these that control the core. The core in debug runs in a step by step mode. Debug TAP has access to CP14 via the core (in dependence upon an access control signal input on the JSDAEN pin shown as JADI pin, JTAG ACCESS DISABLE INPUT in Figure 45) and the control values can also be reset in this way.

15

Access to the CP14 register via debug TAP 580 is controlled by an access control signal JSDAEN. This is arranged so that in order for access and in particular write access to be allowed JSDAEN must be set high. During board stage when the whole processor is being verified, JSDAEN is set high and debug is enabled on the whole system. Once the system has been checked, the JSDAEN pin can be tied to ground, this means that access to the control values that enable debug in secure mode is now not available via Debug TAP 580. Generally processors in production mode have JSDAEN tied to ground. Access to the control values is thus, only available via the software route via instruction memory 550. Access via this route is limited to secure supervisor mode or to another mode provided an authentication code is given (see Figure 68).

20

25

It should be noted that by default debug (intrusive and observable – trace) are only available in non-secure world. To enable them to be available in secure world the control value enable bits need to be set.

30

secure supervisor mode can be entered using an authentication code and then the control value can be set in CP14.

Control logic 620 outputs an “enter debug” signal when address comparator 610 indicates that a breakpoint has been reached provided thread comparator 640 shows that debug is allowable for that thread. This assumes that the thread aware initialisation bit is set in CP14. If the thread aware initialisation bit is set following a breakpoint, debug or trace can only be entered if address and context identifiers match those indicated in the breakpoint and in the allowable thread indicator. Following initiation of a monitoring function, the capture of diagnostic data will only continue while the context identifier is detected by comparator 640 as an allowed thread. When a context identifier shows that the application running is not an allowed one, then the capture of diagnostic data is suppressed.

It should be noted that in the preferred embodiment, there is some hierarchy within the granularity. In effect the secure debug or trace enable bit is at the top, followed by the secure user-mode enable bit and lastly comes the secure thread aware enable bit. This is illustrated in Figures 69A and 69B (see below).

The control values held in the “Debug and Status Control” register (CP14) control secure debug granularity according to the domain, the mode and the executing thread. It is on top of secure supervisor mode. Once the “Debug and Status Control” register CP14 is configured, it’s up to secure supervisor mode to program the corresponding breakpoints, watchpoints, etc to make the core enter Debug State.

Figure 69A shows a summary of the secure debug granularity for intrusive debug. Default values at reset are represented in grey colour.

It is the same for debug granularity concerning observable debug. Figure 69B shows a summary of secure debug granularity in this case, here default values at reset are also represented in grey colour.



In the above embodiment, if secure domain is not enabled, a SMI instruction is always seen as an atomic event and the capture of diagnostic data is suppressed.

Furthermore, if the thread aware initialisation bit is set then granularity of the  
5 monitoring function during operation with respect to application also occurs.

With regard to observable debug or trace, this is done by ETM and is entirely independent of debug. When trace is enabled ETM works as usual and when it is disabled, ETM hides trace in the secure world, or part of the secure world depending  
10 on the granularity chosen. One way to avoid ETM capturing and tracing diagnostic data in the secure domain when this is not enabled is to stall ETM when the S-bit is high. This can be done by combining the S-bit with the ETMPWRDOWN signal, so that the ETM values are held at their last values when the core enters secure world. The ETM should thus trace a SMI instruction and then be stalled until the core returns  
15 to non-secure world. Thus, the ETM would only see non-secure activity.

A summary of some of the different monitoring functions and their granularity is given below.

#### 20 **Intrusive debug at board stage**

At board stage when the JSDAEN pin is not tied, there is the ability to enable debug everywhere before starting any boot session. Similarly, if we are in secure supervisor mode we have similar rights.

25 If we initialise debug in halt debug mode all registers are accessible (non-secure and secure register banks) and the whole memory can be dumped, except the bits dedicated to control debug.

Debug halt mode can be entered from whatever mode and from whatever  
30 domain. Breakpoints and watchpoints can be set in secure or in non-secure memory. In debug state, it is possible to enter secure world by simply changing the S bit via an MCR instruction.

External debug request or internal debug request is taken into account in non-secure world only. If EDBGREQ (external debug request) is asserted while in secure world, the core enters debug halt mode once secure function is terminated and the core is returned in non-secure world.

5

Programming a breakpoint or watchpoint on secure memory has no effect and the core is not stopped when the programmed address matches.

Vector Trap Register (details of this are given below) concerns non-secure exceptions only. All extended trapping enable bits explained before have no effect.

10

Once in halt debug mode the following restrictions apply:

S bit cannot be changed to force secure world entry, unless secure debug is enabled.

15

Mode bits can not be changed if debug is permitted in secure supervisor mode only.

Dedicated bits that control secure debug cannot be changed.

If a SMI is loaded and executed (with system speed access), the core re-enters debug state only when secure function is completely executed.

20

In monitor debug mode because monitoring cannot occur in secure world, the secure abort handler does not need to support a debug monitor programme. In non secure world, step-by-step is possible but whenever an SMI is executed secure function is executed entirely in other words an XWSI only “step-over” is allowed while “step-in” and “step-over” are possible on all other instructions. XWSI is thus considered an atomic instruction.

25

Once secure debug is disabled, we have the following restrictions:

30

Before entering monitor mode:

In summary, in debug, mode bits can be altered only if debug is enabled in secure supervisor mode. It will prevent anybody that has access to debug in the secure domain to have access to all secure world by altering the system (modifying TBL entries, etc). In that way each thread can debug its own code, and only its own code. The secure kernel must be kept safe. Thus when entering debug while the core is running in non-secure world, mode bits can only be altered as before.

Embodiments of the technique use a new **vector trap register**. If one of the bits in this register is set high and the corresponding vector triggers, the processor enters debug state as if a breakpoint has been set on an instruction fetch from the relevant exception vector. The behaviour of these bits may be different according to the value of 'Debug in Secure world Enable' bit in debug control register.

The new vector trap register comprises the following bits: D\_s\_abort, P\_s\_abort, S\_undef, SMI, FIQ, IRQ, Unaligned, D\_abort, P\_abort, SWI and Undef.

- D\_s\_abort bit: should only be set when debug is enabled in secure world and when debug is configured in halt debug mode. In monitor debug mode, this bit should never bit set. If debug in secure world is disabled, this bit has no effect whatever its value.
- P\_s\_abort bit: same as D\_s\_abort bit.
- S\_undef bit: should only be set when debug is enable in secure world. If debug in secure world is disabled, this bit has no effect whatever its value is.
- SMI bit: should only be set when debug is enabled in secure world. If debug in secure world is disabled, this bit has no effect whatever its value is.
- FIQ, IRQ, D\_abort, P\_abort, SWI, undef bits: correspond to non-secure exceptions, so they are valid even if debug in secure world is disabled. Note that D\_abort and P\_abort should not be asserted high in monitor mode.
- Reset bit: as we enter secure world when reset occurs, this bit is valid only when debug in secure world is enabled, otherwise it has no effect.

## CLAIMS

1. A data processing apparatus, comprising:

5 a processor operable in a plurality of modes and a plurality of domains, said plurality of domains comprising a secure domain and a non-secure domain, said plurality of modes including at least one non-secure mode being a mode in the non-secure domain, and at least one secure mode being a mode in the secure domain, said processor being operable such that when executing a program in a secure mode said  
10 program has access to secure data which is not accessible when said processor is operating in a non-secure mode;

a memory operable to store data required by the processor and comprising secure memory for storing secure data and non-secure memory for storing non-secure data, the processor being operable to issue a memory access request when access to an item of  
15 data in the memory is required;

at least one memory management unit operable, upon receipt of the memory access request from the processor, to perform conversion of a virtual address specified by the memory access request to a physical address;

a first set of tables, each table in the first set containing a number of first  
20 descriptors, each first descriptor containing at least a virtual address portion and a corresponding intermediate address portion;

a second set of tables, each table in the second set containing a number of second descriptors, each second descriptor containing at least an intermediate address portion and a corresponding physical address portion, the second set of tables being managed by  
25 the processor when operating in a privileged mode which is not a non-secure mode;

the at least one memory management unit being operable to cause predetermined tables in said first and second set to be referenced to enable the conversion of the virtual address specified by the memory access request to a physical address.

30 2. A data processing apparatus as claimed in Claim 1, wherein the privileged mode is a monitor mode in which the processor is operable to manage switching between said secure domain and said non-secure domain.

for that first descriptor, the second memory management unit being operable to receive the table lookup request and determine the physical address corresponding to that intermediate address.

5     10.     A data processing apparatus as claimed in Claim 9, wherein the second memory management unit is then operable to cause the first descriptor at that physical address to be retrieved and returned to the first memory management unit.

10     11.     A data processing apparatus as claimed in any of claims 7 to 10, wherein the first memory management unit comprises a first internal storage unit for storing first descriptors retrieved from the predetermined table of the first set, and used by the first memory management unit to derive access control information used to perform the conversion of the virtual address into a corresponding intermediate address.

15     12.     A data processing apparatus as claimed in Claim 11, wherein the first internal storage unit is a first translation lookaside buffer (TLB) operable to store the first descriptors retrieved from the predetermined table of the first set.

20     13.     A data processing apparatus as claimed in claim 12, wherein the first TLB is a first main TLB for storing the first descriptors retrieved by the first memory management unit from the predetermined table of the first set, and the internal storage further comprises a micro-TLB for storing the access control information derived from the first descriptors, the access control information comprising conversions between a number of virtual address portions and corresponding intermediate address portions, and  
25     the access control information being transferred from the first main TLB to the micro-TLB prior to use of that access control information by the first memory management unit.

30     14.     A data processing apparatus as claimed in any of claims 7 to 10, wherein the first memory management unit comprises a first internal storage unit for storing new descriptors derived from corresponding first and second descriptors retrieved from the predetermined tables of the first and second sets, and used by the first memory

conversions between a number of intermediate address portions and corresponding physical address portions, and the access control information being transferred from the second main TLB to the micro-TLB prior to use of that access control information by the second memory management unit.

5

20. A data processing apparatus as claimed in Claim 18 or Claim 19, when dependent on Claim 12 or Claim 15, wherein the first and second sets of tables each comprise at least a secure table and a non-secure table, the first and second TLBs comprising a flag associated with each descriptor stored therein to identify whether that  
10 descriptor is derived from said non-secure table or said secure table.

21. A data processing apparatus as claimed in Claim 20 when dependent on Claim 19 and one of Claims 13 or 16, wherein the micro-TLB of both the first and second memory management units is flushed whenever the mode of operation of the processor  
15 changes between a secure mode and a non-secure mode, in the secure mode access control information only being transferred to the micro-TLB from a descriptor in the associated first or second main TLB that said associated flag indicates is from the secure table, and in the non-secure mode access control information only being transferred to the micro-TLB from a descriptor in the associated first or second main TLB that said  
20 associated flag indicates is from the non-secure table.

22. A data processing apparatus as claimed in any of claims 1 to 6, wherein the at least one memory management unit comprises a single memory management unit, and the processor is operable to execute table merging code to reference the predetermined  
25 tables of the first and second sets in order to produce from a first descriptor and an associated second descriptor a new descriptor associating a virtual address portion with a corresponding physical address portion.

23. A data processing apparatus as claimed in Claim 22, wherein the table merging code is operable to retrieve the first descriptor after referencing the predetermined table  
30 in the second set to obtain the physical address of the first descriptor.

29. A data processing apparatus as claimed in Claim 27 or Claim 28, wherein the first and second sets of tables each comprise at least a secure table and a non-secure table, the TLB comprising a flag associated with each new descriptor stored therein to identify whether that new descriptor is derived from said non-secure tables or said  
5 secure tables.

30. A data processing apparatus as claimed in Claim 29 when dependent on Claim 28, wherein the micro-TLB of the single memory management unit is flushed whenever the mode of operation of the processor changes between a secure mode and a non-secure  
10 mode, in the secure mode access control information only being transferred to the micro-TLB from a new descriptor in the main TLB that said associated flag indicates is derived from secure tables, and in the non-secure mode access control information only being transferred to the micro-TLB from a new descriptor in the main TLB that said associated flag indicates is derived from non-secure tables.

15

31. A data processing apparatus as claimed in any of claims 22 to 30 when dependent on Claim 2, wherein the table merging code is executed by the processor when operating in the monitor mode.

20 32. A data processing apparatus as claimed in any preceding claim, wherein said first and second sets of tables comprise page tables.

33. A data processing apparatus as claimed in any preceding claim, wherein the first set of tables and the second set of tables are stored within said memory.

25

34. A method of controlling access to a memory in a data processing apparatus, the data processing apparatus comprising a processor operable in a plurality of modes and a plurality of domains, said plurality of domains comprising a secure domain and a non-secure domain, said plurality of modes including at least one non-secure mode  
30 being a mode in the non-secure domain, and at least one secure mode being a mode in the secure domain, said processor being operable such that when executing a program in a secure mode said program has access to secure data which is not accessible when

select the predetermined tables in said first and second sets, dependent on whether the domain being switched to is the secure domain or the non-secure domain.

39. A method as claimed in any of claims 34 to 37, wherein the predetermined tables  
5 within said first and second sets are selected when the tables are to be referenced dependent on whether the processor is operating in a secure mode or a non-secure mode at the time the memory access request is issued.

40. A method as claimed in any of claims 34 to 39, wherein said step of performing  
10 conversion of a virtual address to a physical address is performed by at least one of a first memory management unit and a second memory management unit.

41. A method as claimed in Claim 40, wherein if the first memory management unit  
needs to access a first descriptor within a predetermined table of said first set, the  
15 method further comprises the steps of:

issuing from the first memory management unit a table lookup request  
specifying an intermediate address for that first descriptor; and

receiving the table lookup request at the second memory management unit and  
determining the physical address corresponding to that intermediate address.

20

42. A method as claimed in Claim 41, further comprising the step of:  
causing the first descriptor at that physical address to be retrieved and returned to  
the first memory management unit.

25 43. A method as claimed in Claim 42, further comprising the steps of:  
causing a second descriptor within a predetermined table of said second set to be  
retrieved; and

merging the first descriptor and second descriptor in order to produce a new  
descriptor for storing in the first memory management unit, the new descriptor  
30 containing at least a virtual address portion and a corresponding physical address  
portion.



49. A method as claimed in any of claims 44 to 48 when dependent on Claim 35, wherein the table merging code is executed by the processor when operating in the monitor mode.

5 50. A computer program providing table merging code and operable to configure a processor of a data processing apparatus to perform the method of claims 44 to 46.

51. A computer program product carrying a computer program as claimed in Claim 50.

10

52. A data processing apparatus, substantially as hereinbefore described with reference to Figures 50 to 54.

53. A method of controlling access to a memory in a data processing apparatus,  
15 substantially as hereinbefore described with reference to Figures 50 to 54.

1/62

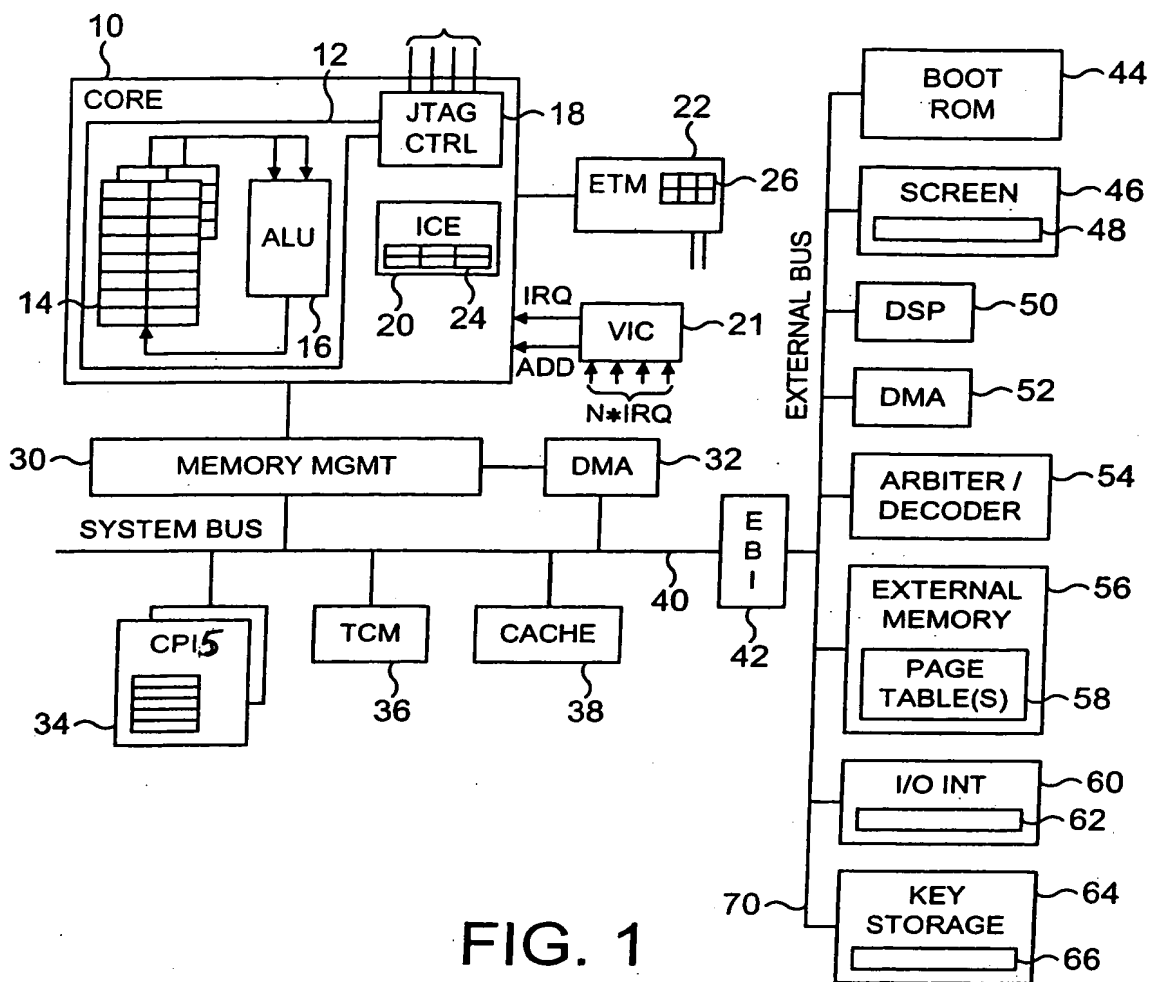


FIG. 1

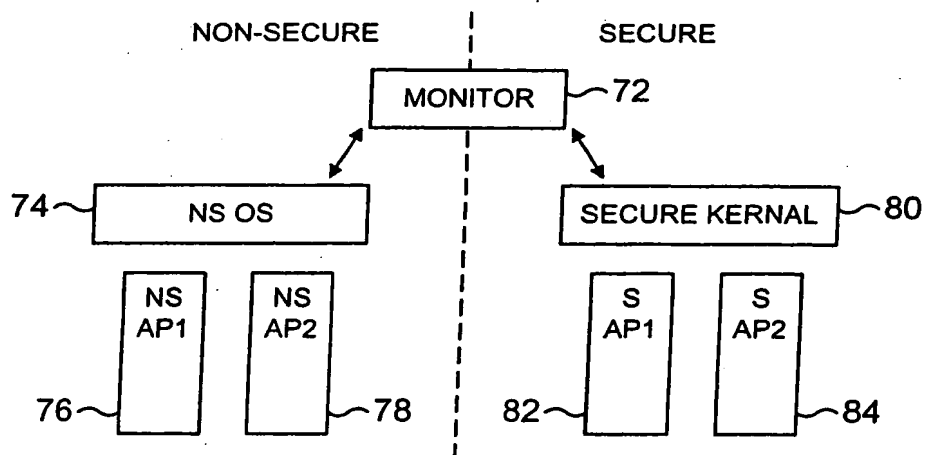


FIG. 2

2/62

		<u>DOMAIN</u>	
		NON-SECURE	SECURE
<u>MODE</u>		MONITOR	
	1	NS MODE 1	S MODE 1
	2	NS MODE 2	S MODE 2

FIG. 3

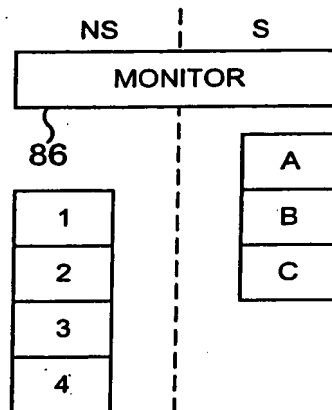


FIG. 4

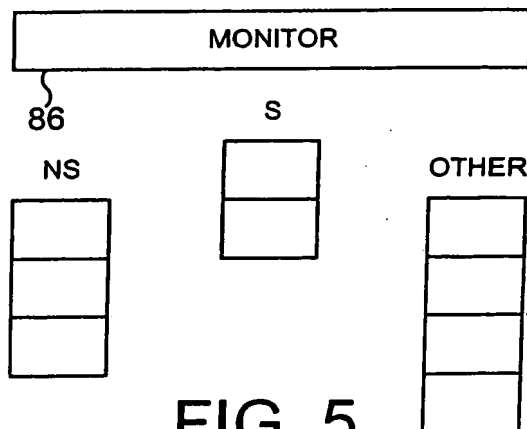


FIG. 5

[illegible]

FIG. 6

4/62

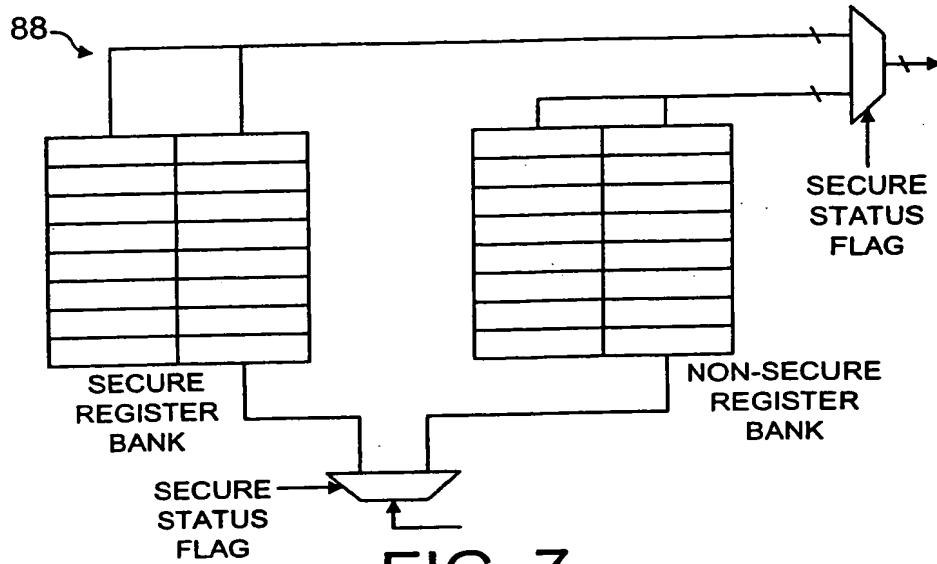


FIG. 7

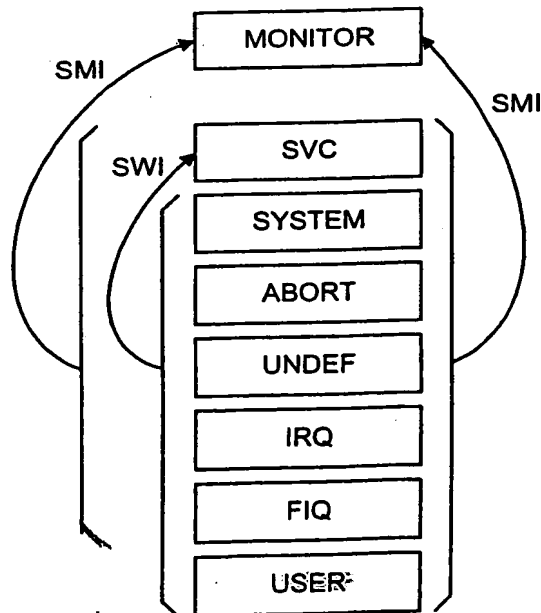


FIG. 8

5/62

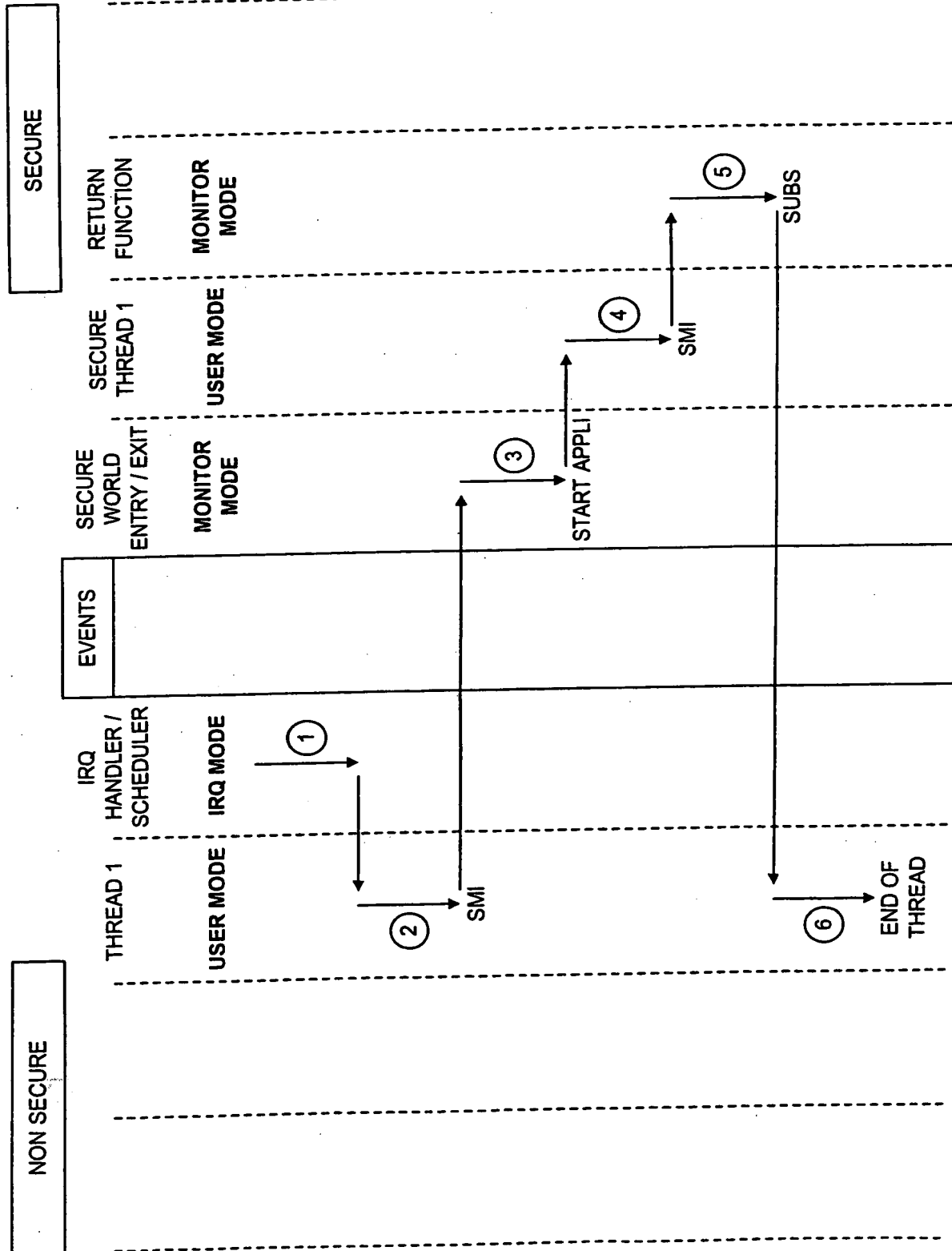


FIG. 9

6/6.2

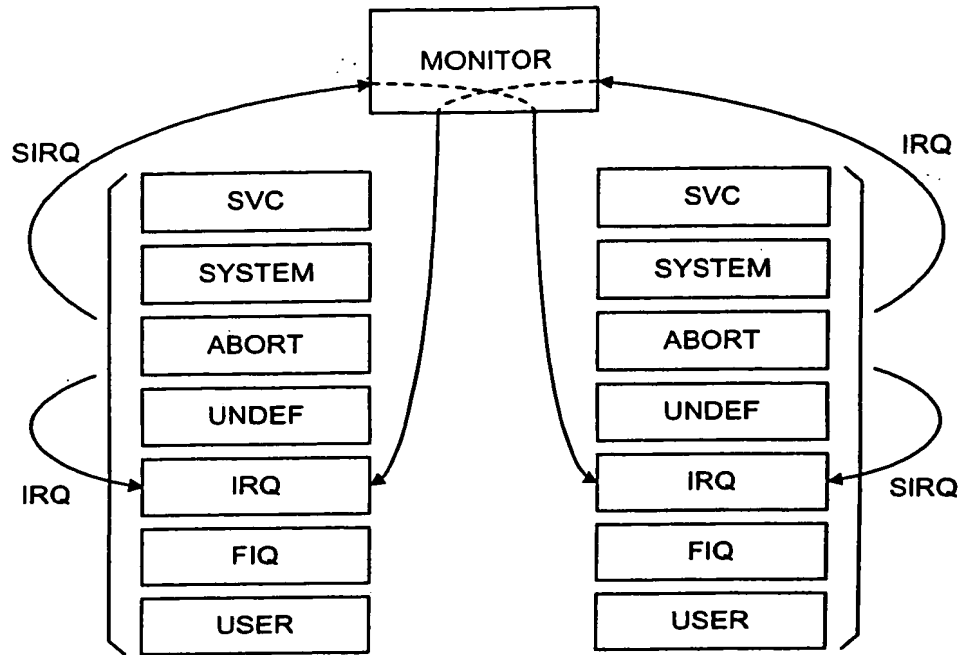


FIG. 10

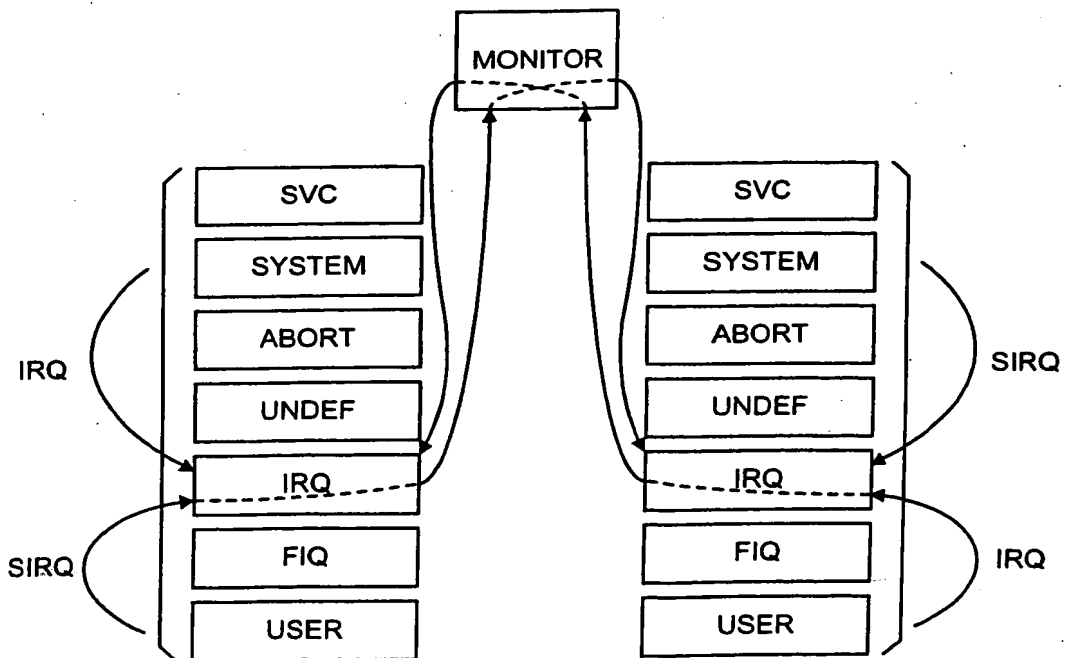


FIG. 12

7/62

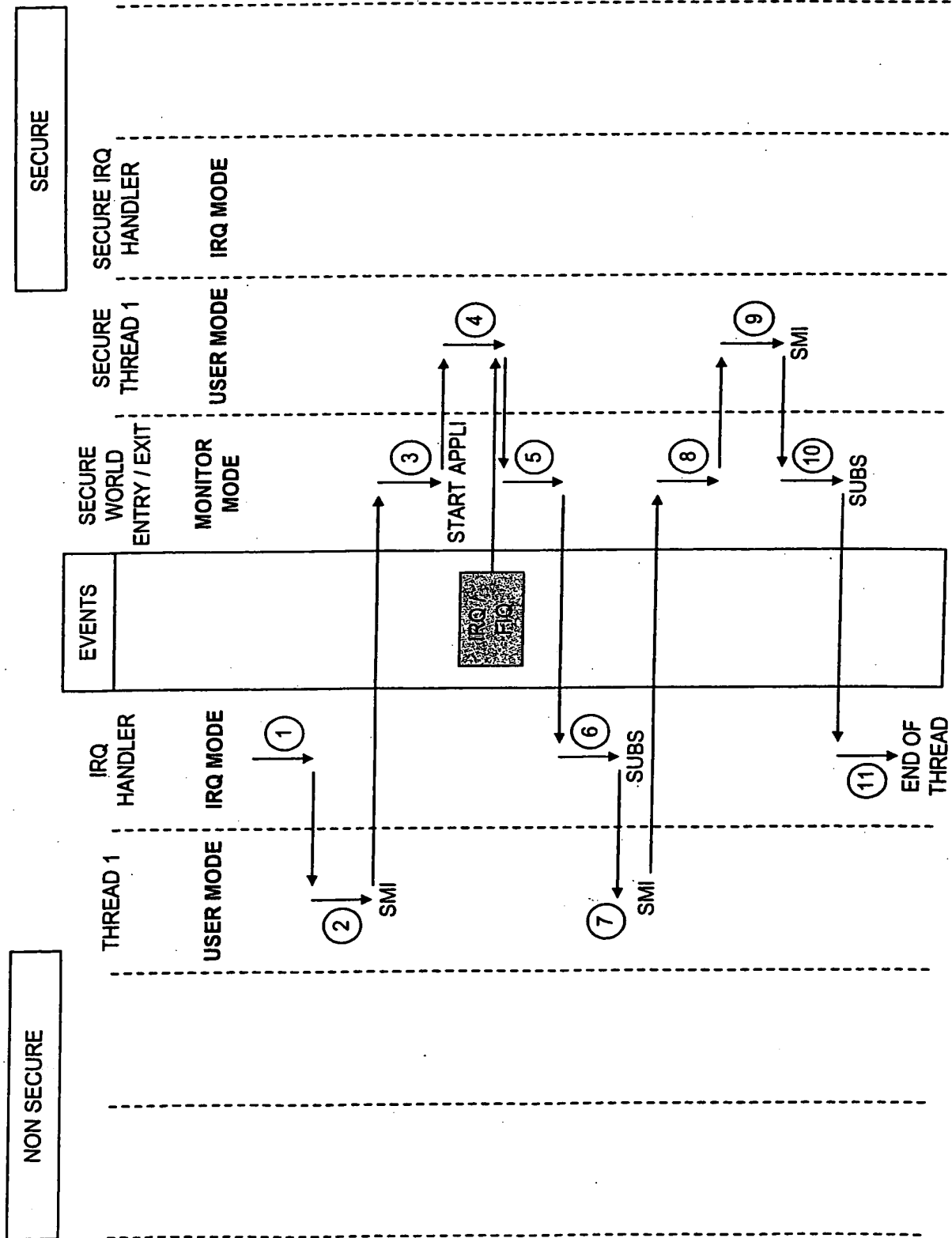


FIG. 11A



8162

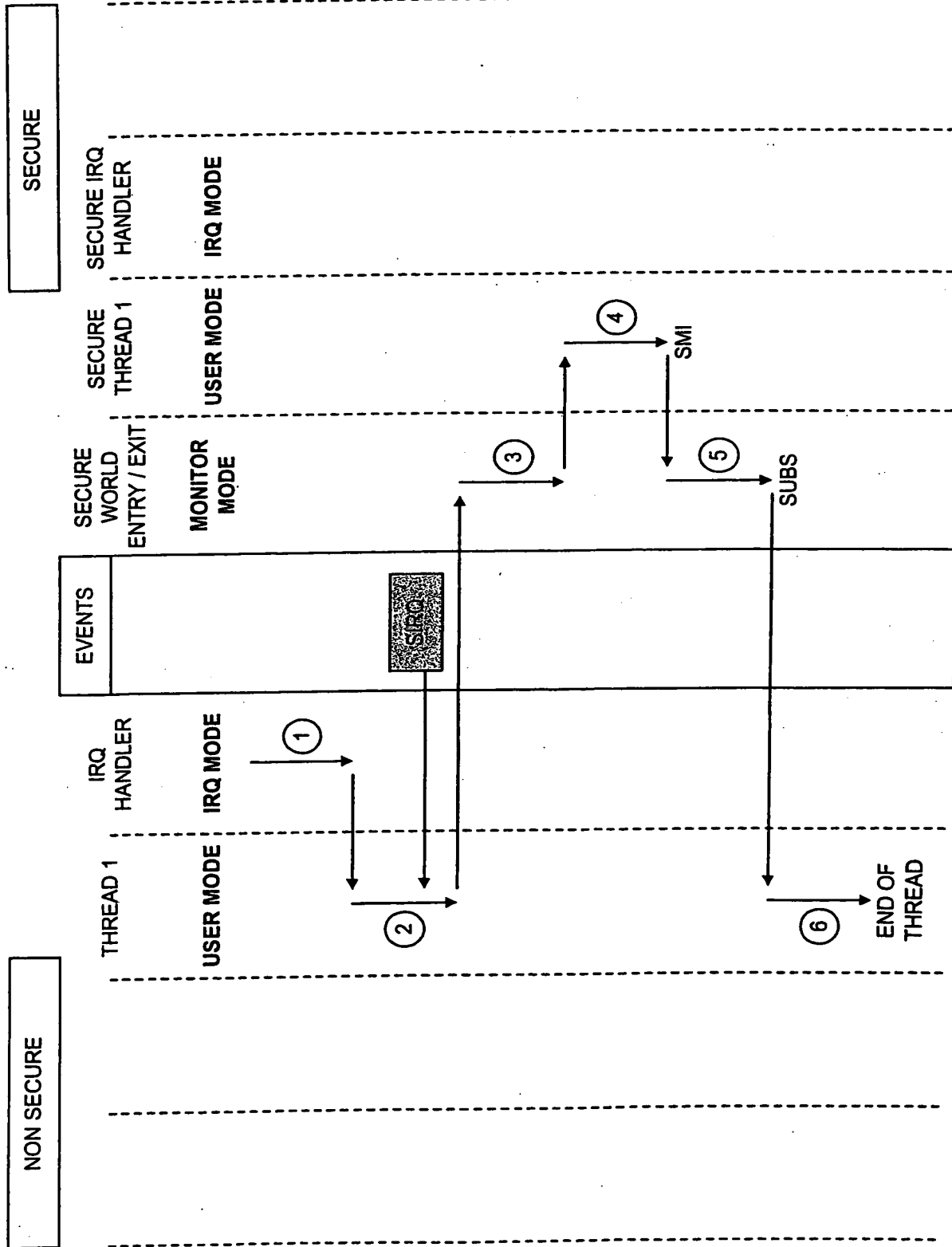


FIG. 11B

9/62

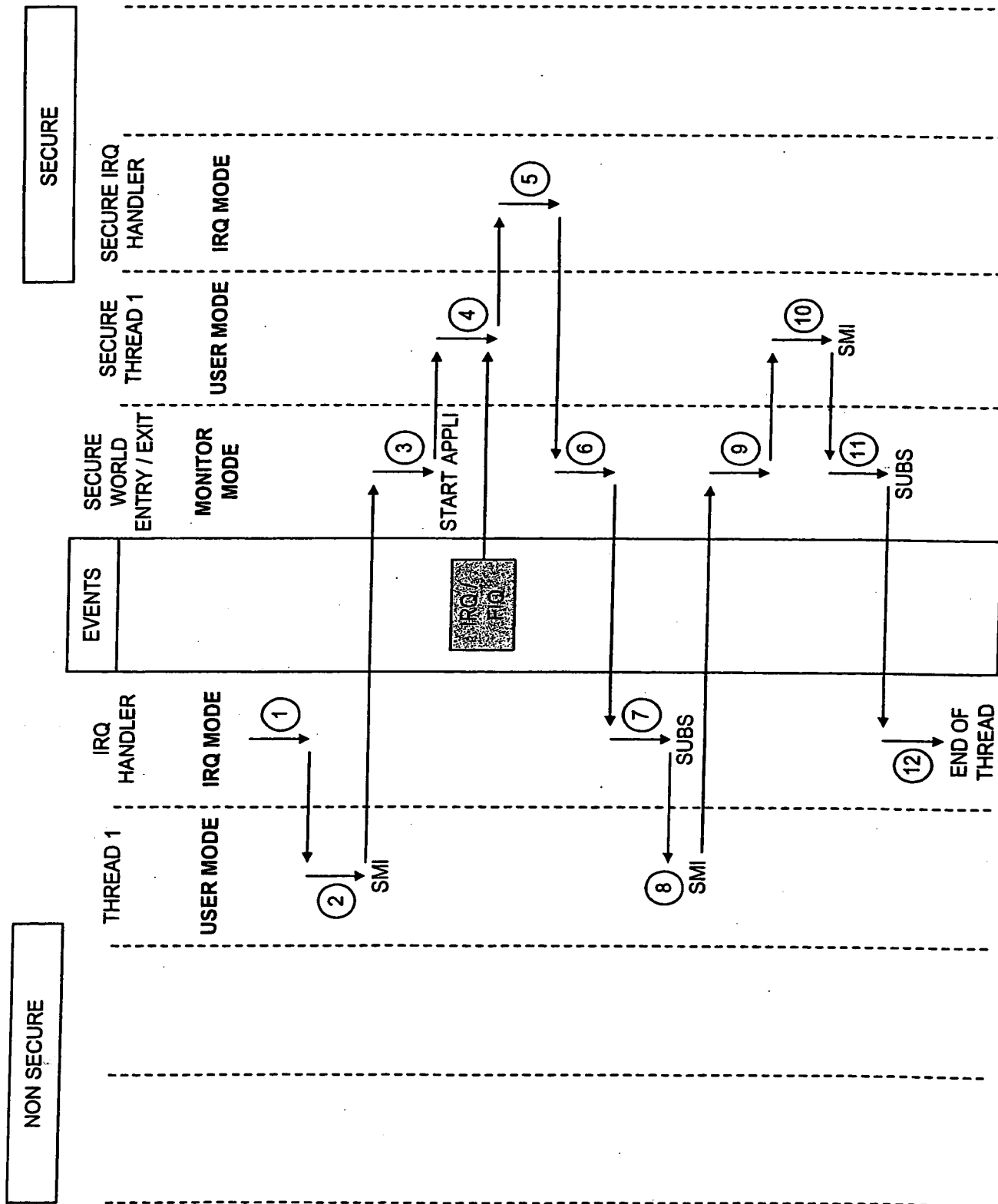


FIG. 13A

[illegible]

FIG. 13B

11/62

EXCEPTION	VECTOR OFFSET	CORRESPONDING MODE
RESET	0x00	SUPERVISOR MODE
UNDEF	0x04	MONITOR MODE / UNDEF MODE
SWI	0x08	SUPERVISOR MODE / MONITOR MODE
PREFETCH ABORT	0x0C	ABORT MODE / MONITOR MODE
DATA ABORT	0x10	ABORT MODE / MONITOR MODE
IRQ / SIRQ	0x18	IRQ MODE / MONITOR MODE
FIQ	0x1C	FIQ MODE / MONITOR MODE
SMI	0x20	UNDEF MODE / MONITOR MODE

FIG. 14

MONITOR	
RESET	VM0
UNDEF	VM1
SWI	VM2
PREFETCH ABORT	VM3
DATA ABORT	VM4
IRQ / SIRQ	VM5
FIQ	VM6
SMI	VM7

SECURE	
RESET	VS0
UNDEF	VS1
SWI	VS2
PREFETCH ABORT	VS3
DATA ABORT	VS4
IRQ / SIRQ	VS5
FIQ	VS6
SMI	VS7

NON-SECURE	
RESET	VNS0
UNDEF	VNS1
SWI	VNS2
PREFETCH ABORT	VNS3
DATA ABORT	VNS4
IRQ / SIRQ	VNS5
FIQ	VNS6
SMI	VNS7

FIG. 15

12/62

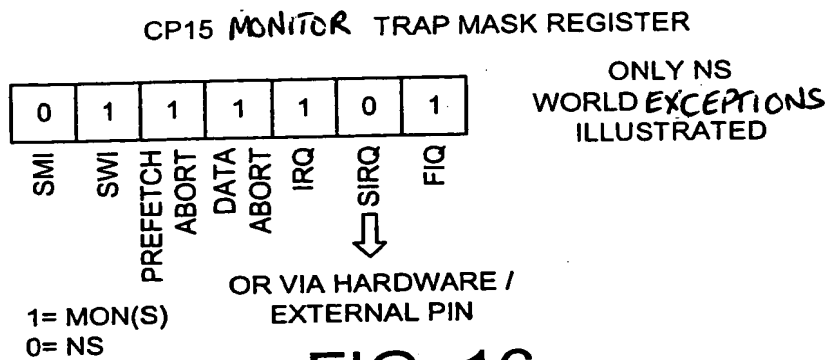


FIG. 16

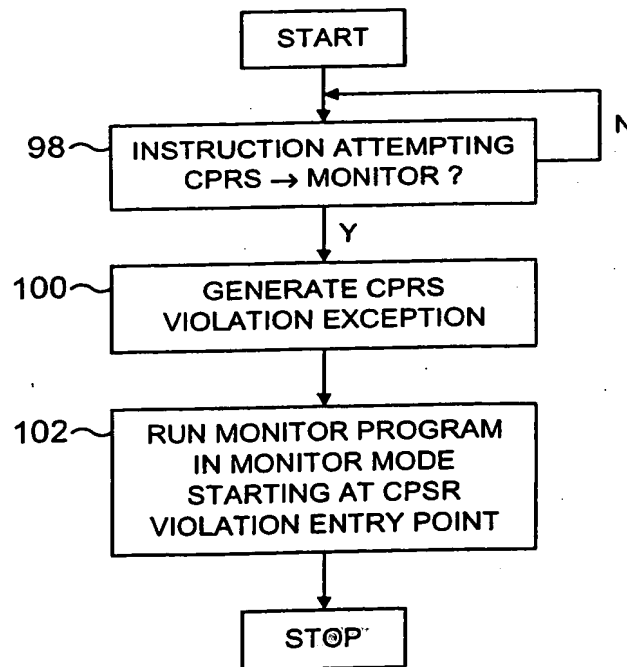


FIG. 17

13/62

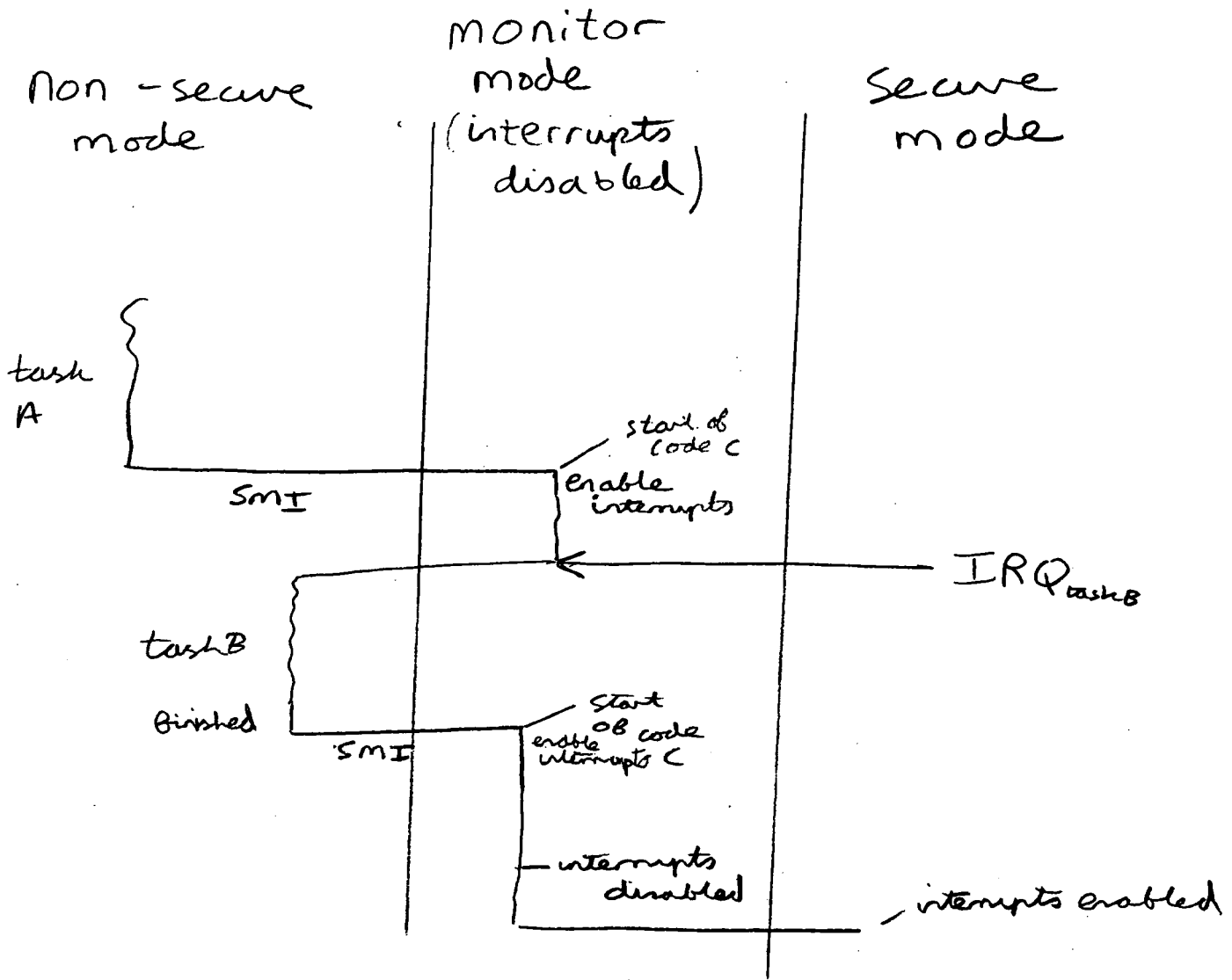


Fig. 18

14/62

non-secure  
mode

monitor  
mode  
(interrupts  
disabled)

secure  
mode

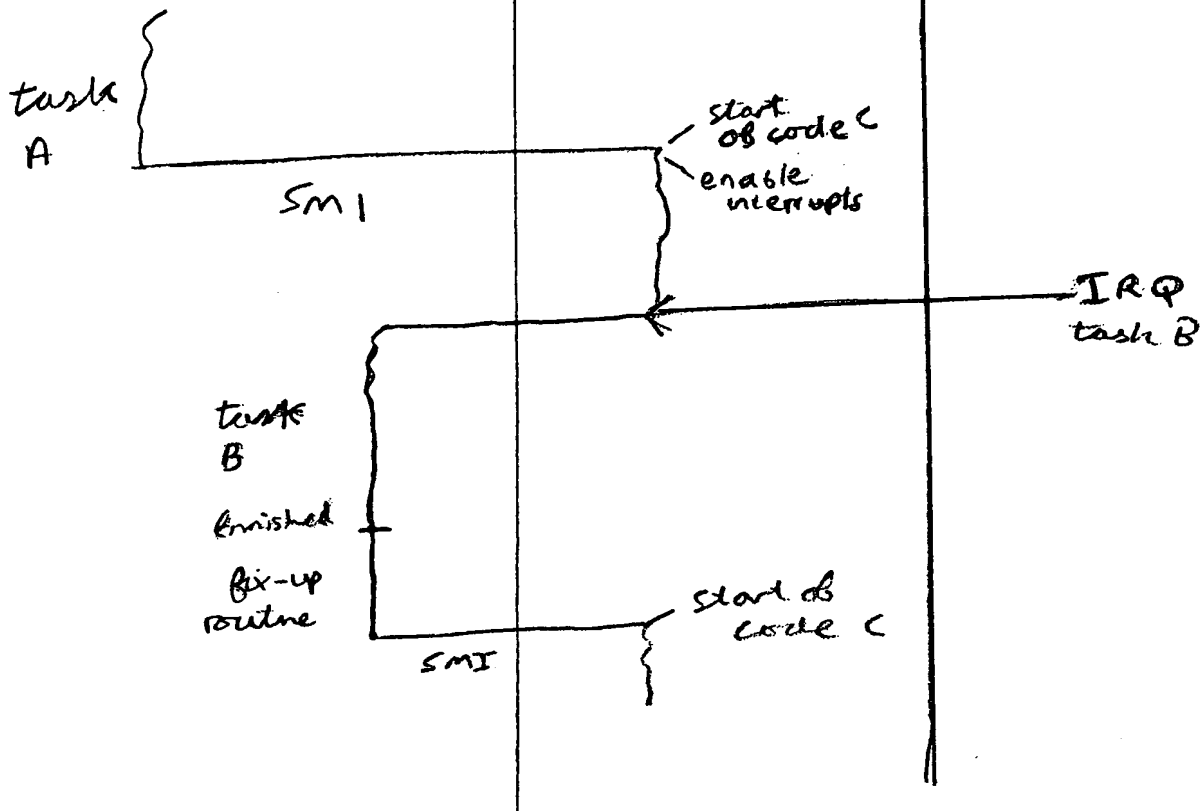


Fig. 19

15/62

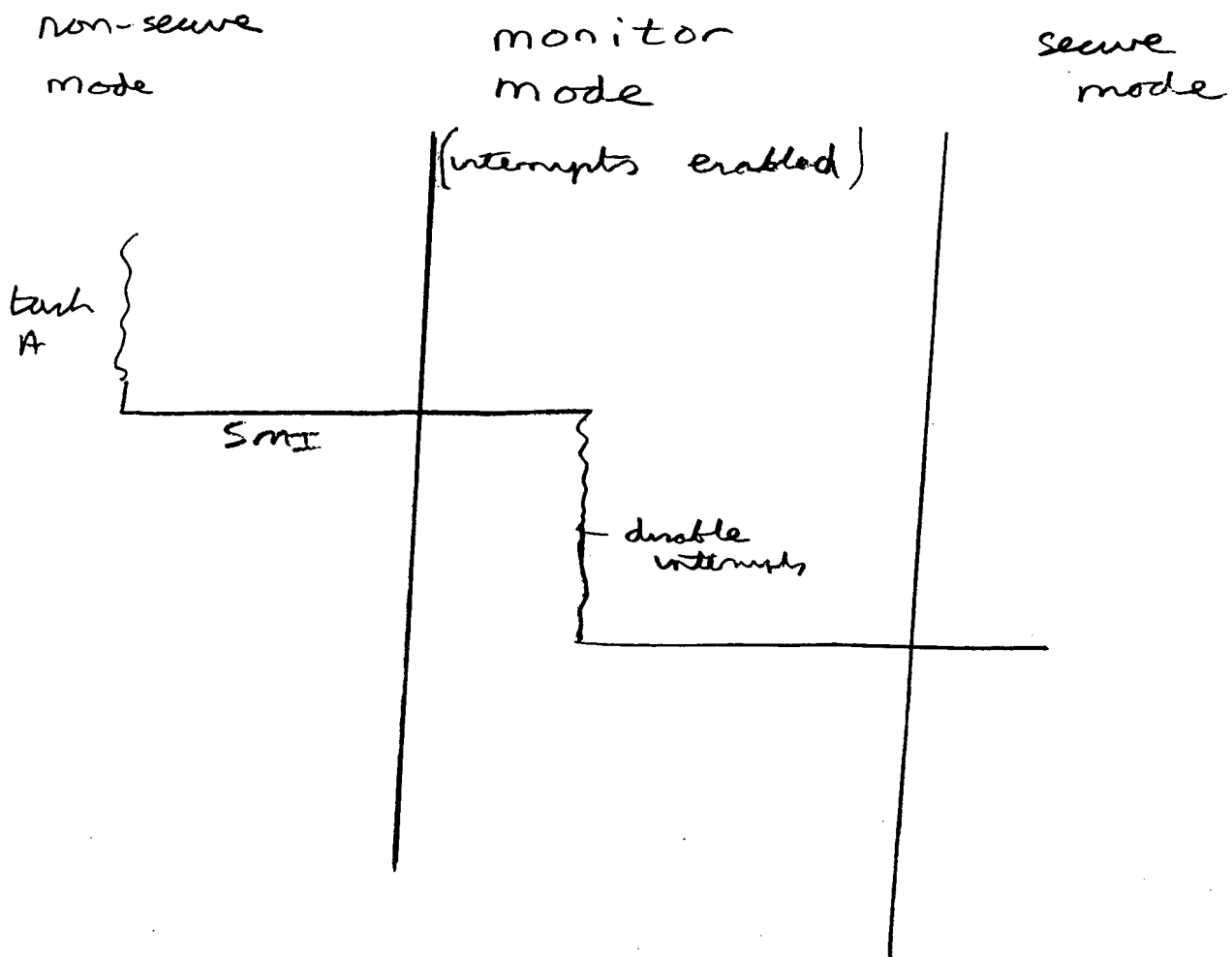


Fig. 20



16/62

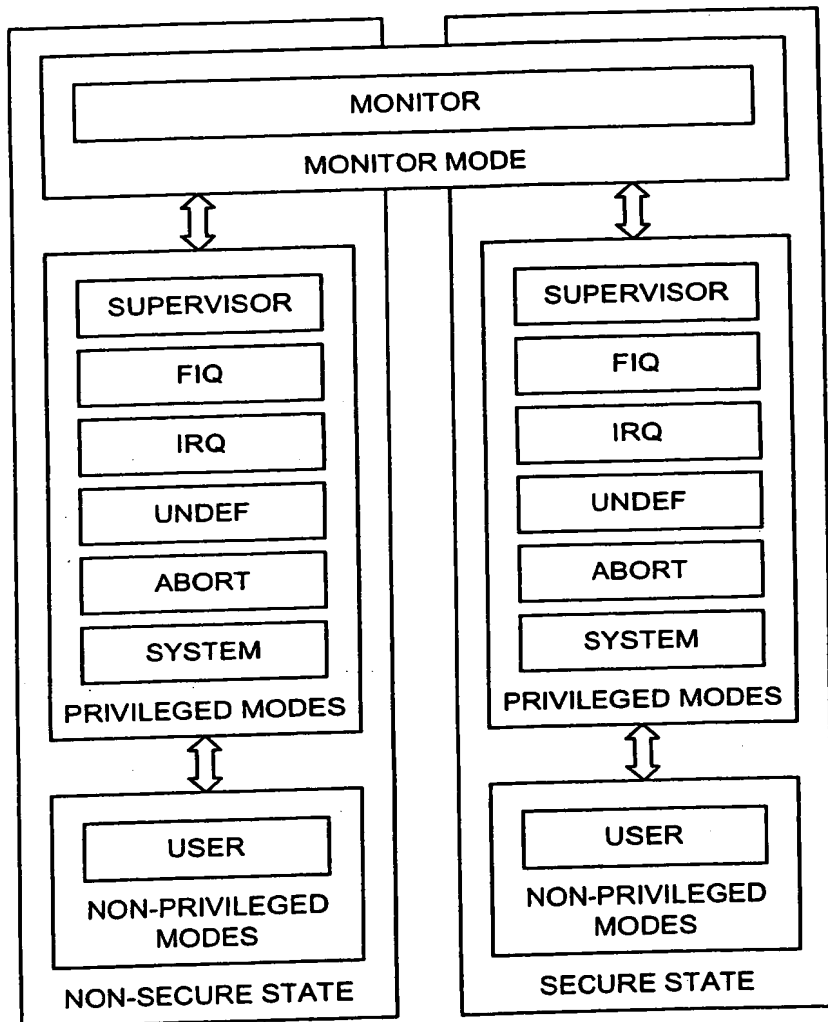


FIG. 21

17/62

USER	SYSTEM	SUPERVISOR	ABORT	UNDEFINED	INTERRUPT	FAST INTERRUPT	MONITOR
R0	R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_FIQ	R8
R9	R9	R9	R9	R9	R9	R9_FIQ	R9
R10	R10	R10	R10	R10	R10	R10_FIQ	R10
R11	R11	R11	R11	R11	R11	R11_FIQ	R11
R12	R12	R12	R12	R12	R12	R12_FIQ	R12
R13	R13	R13_SVC	R13_AB	R13_UND	R13_IRQ	R13_FIQ	R13_MON
R14	R14	R14_SVC	R14_AB	R14_UND	R14_IRQ	R14_FIQ	R14_MON
PC	PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_SVC	SPSR_AB	SPSR_UND	SPSR_IRQ	SPSR_FIQ	SPSR_MON

FIG. 22

18/62

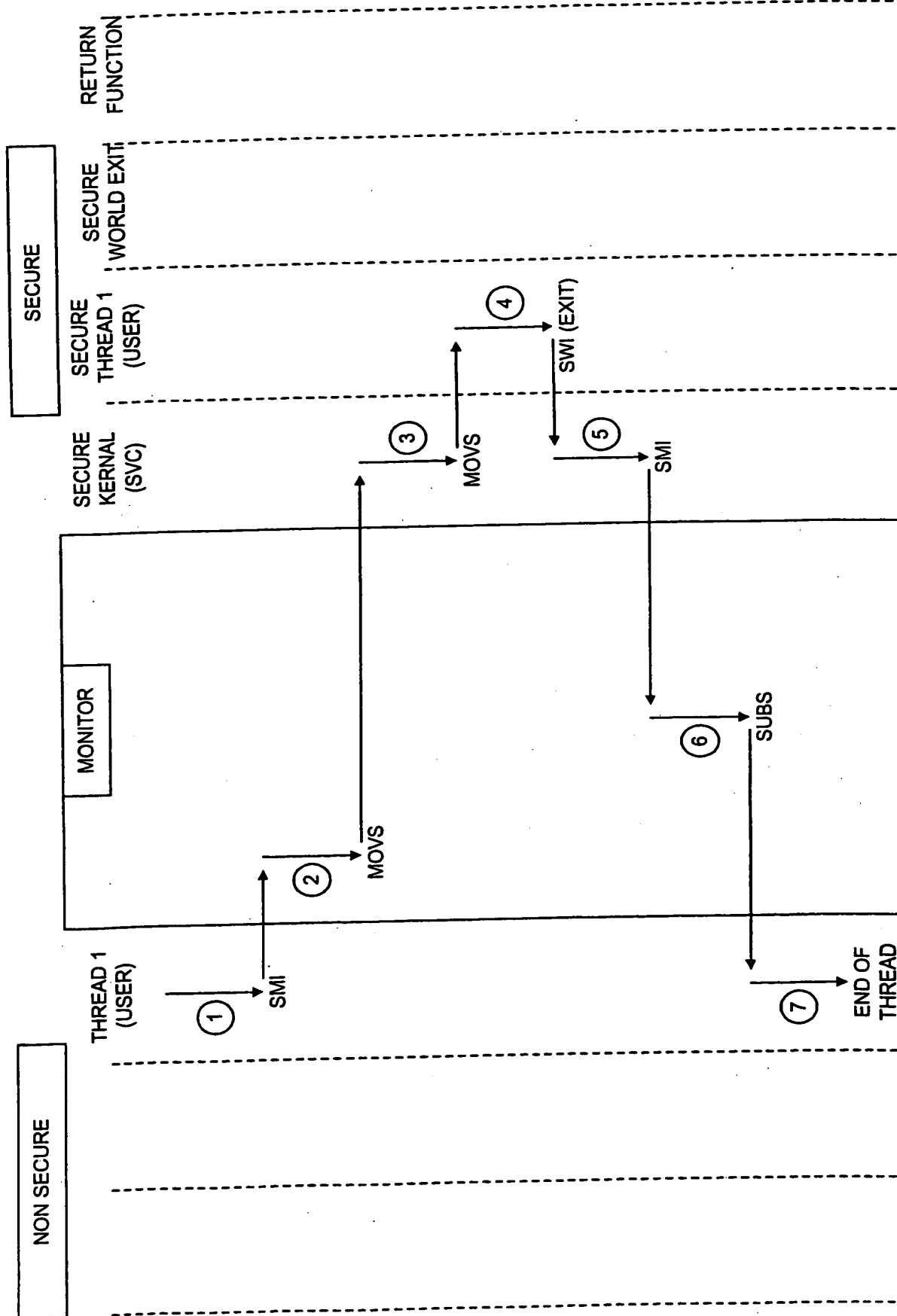
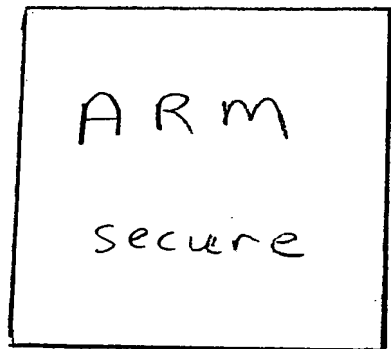
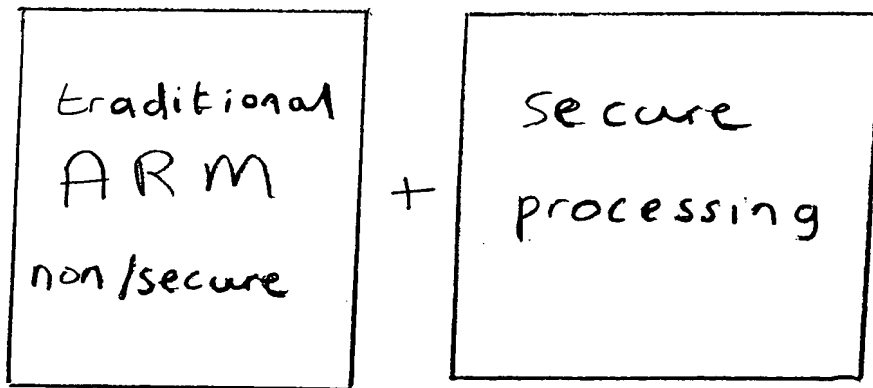


FIG. 23

19/62



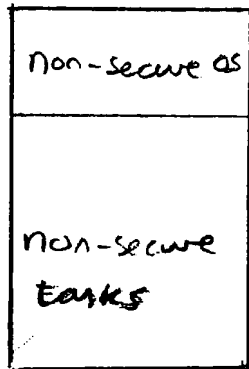
$S = 1$

Fig. 24

20/62

monitor

$S = 1$



$S = 0$

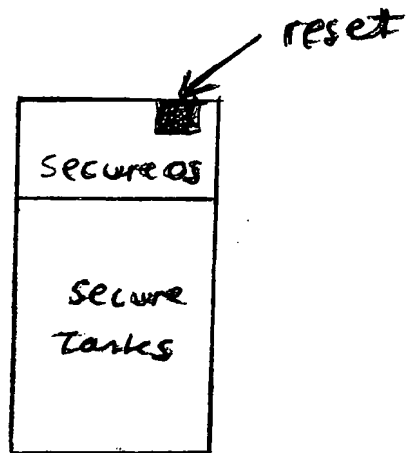


Fig. 25

21/62

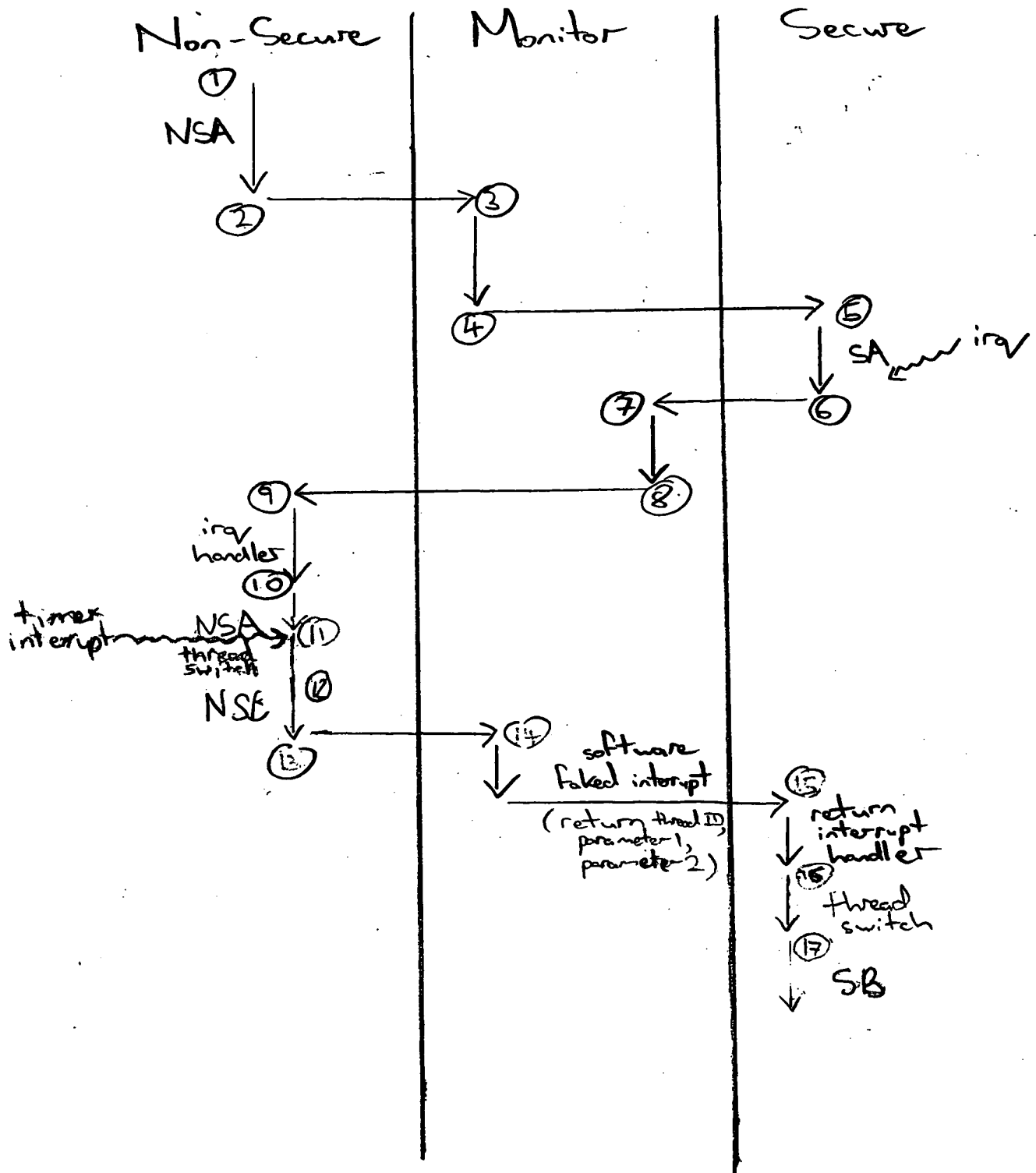


Fig. 26

22/62

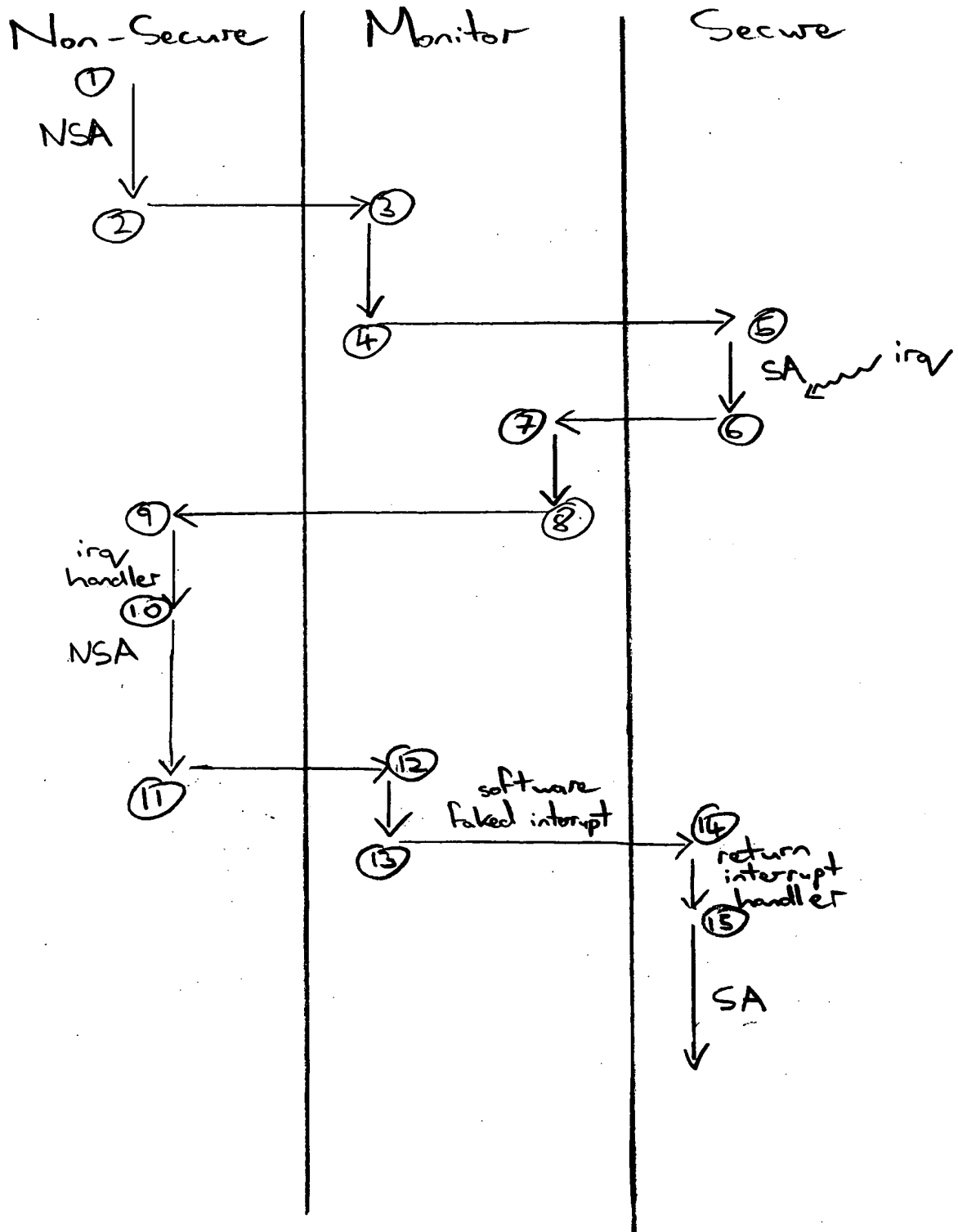


Fig. 27

23/62

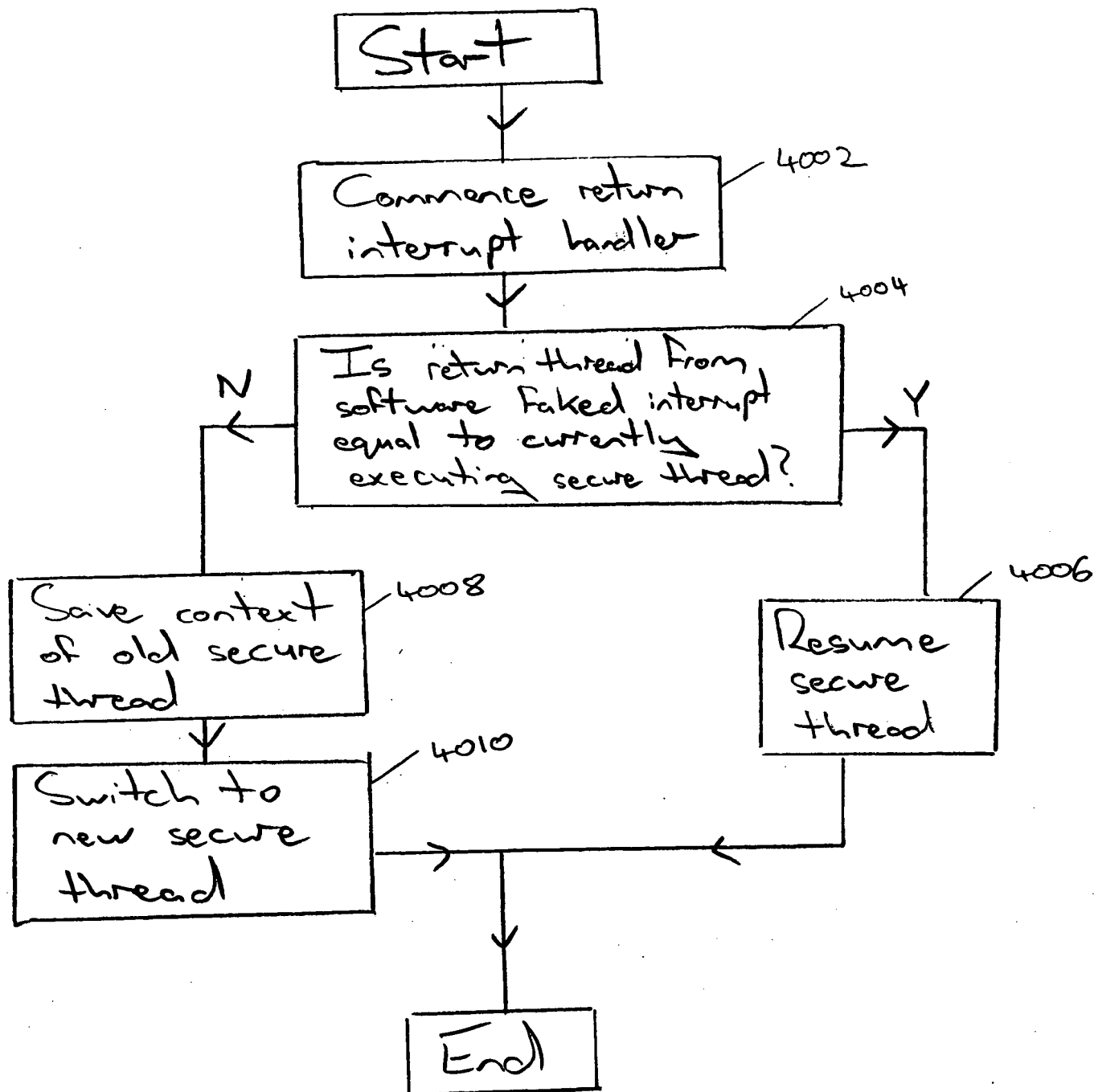


Fig. 28



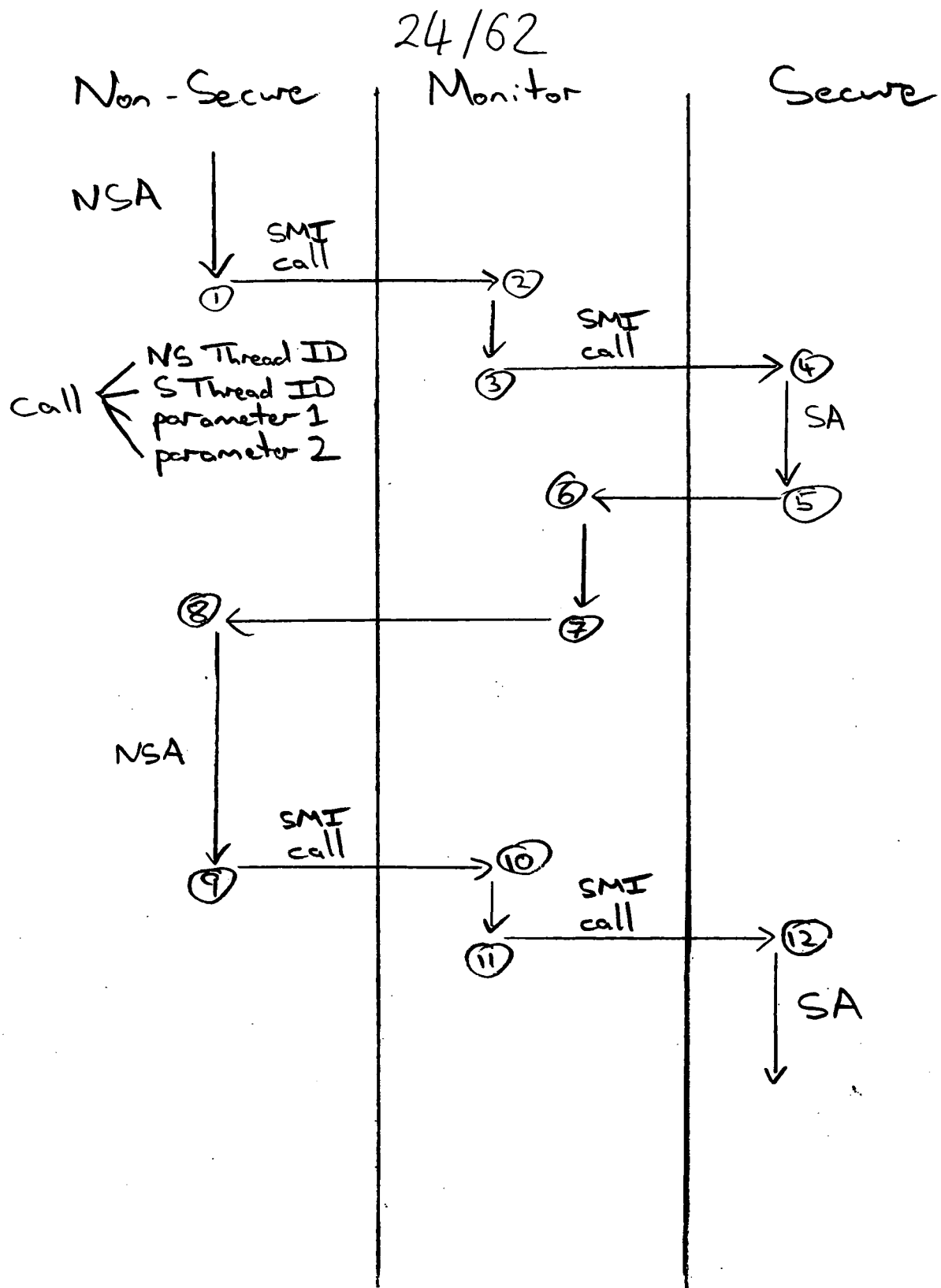


Fig. 29

25/62

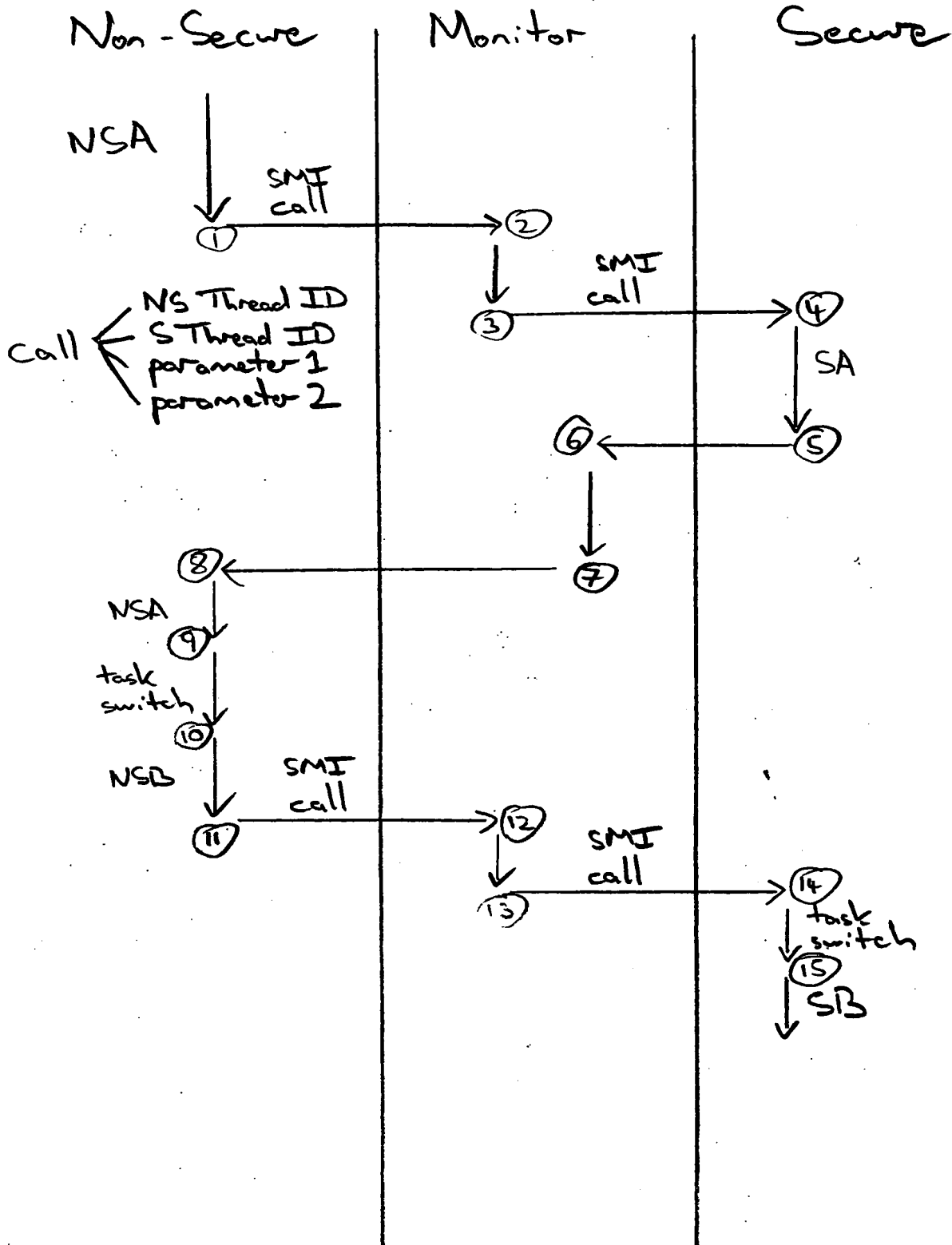


Fig. 30

26/62

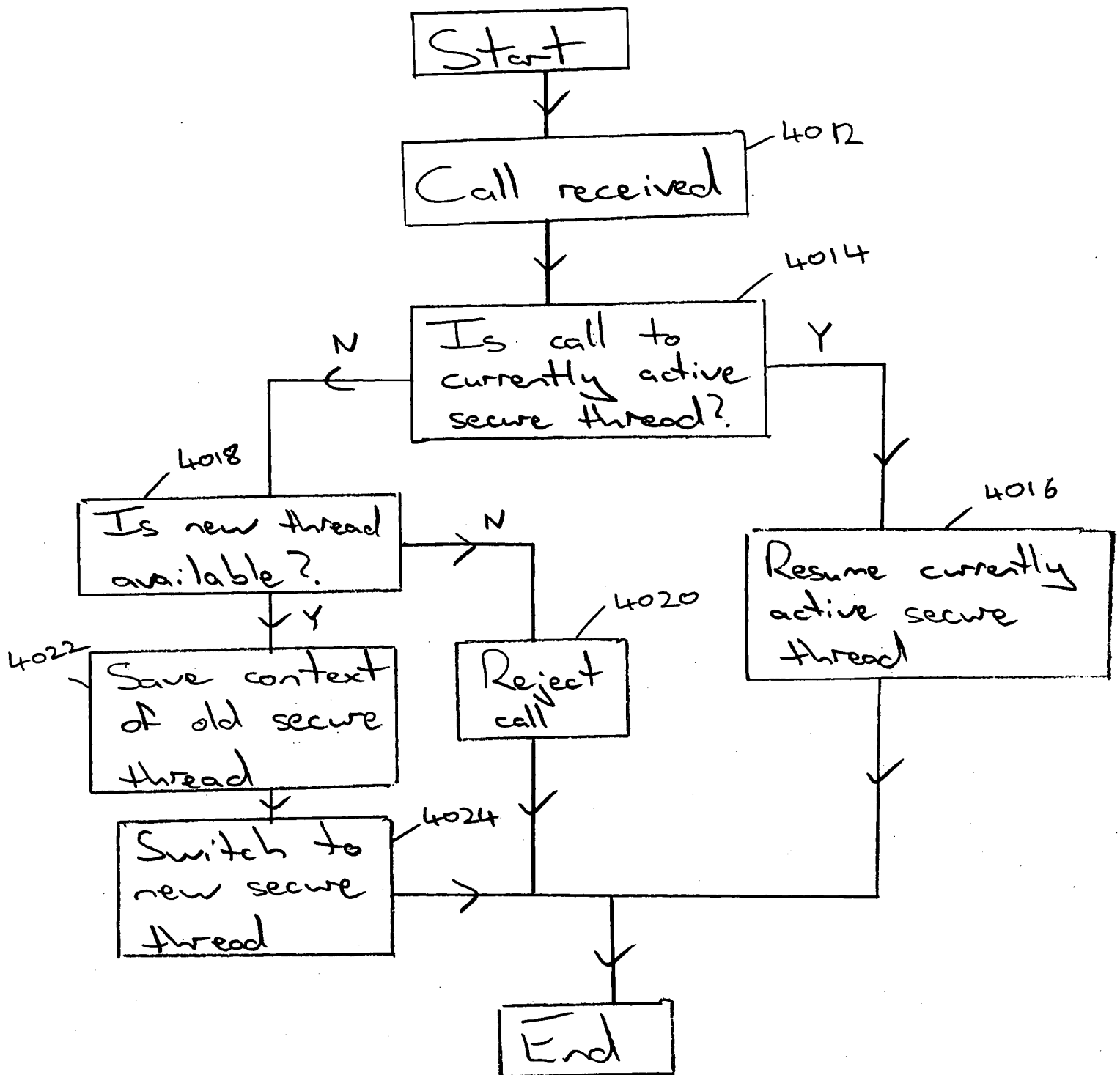


Fig. 31

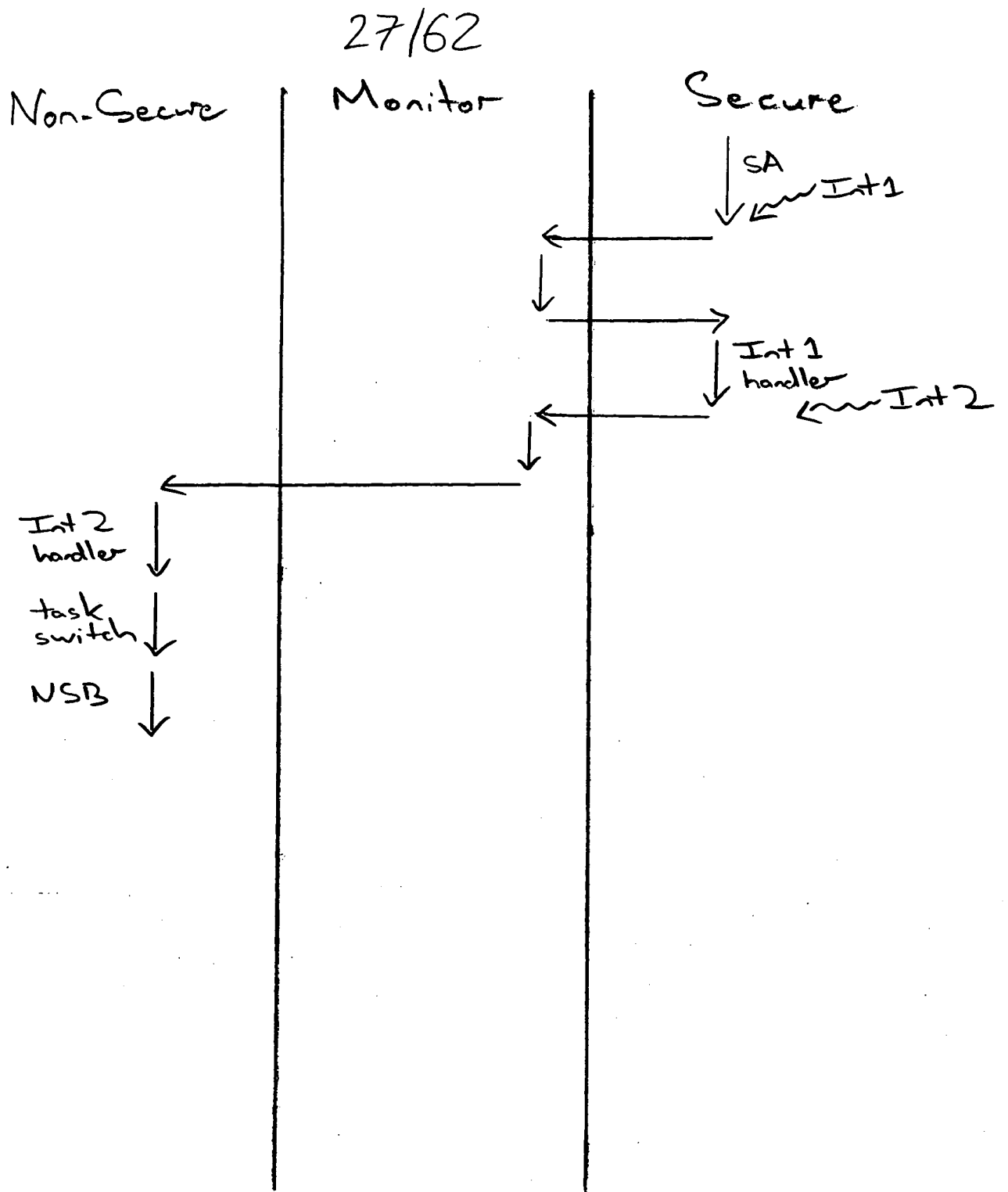


Fig. 32

28/62

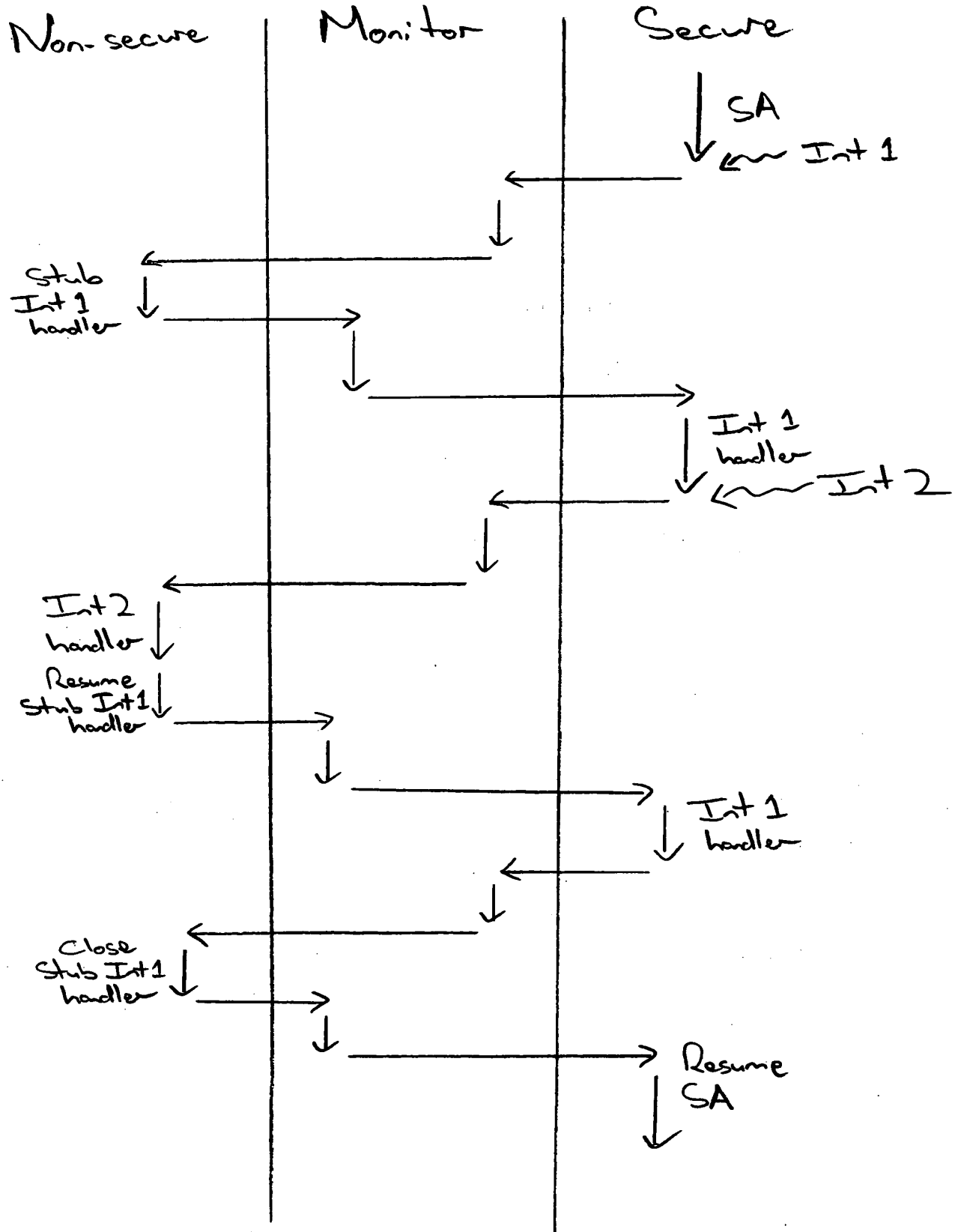


Fig 33

29/62

Interrupt  
Type / Priority

How  
Handled

1

S

2

S

3

NS

4

NS/S

5

NS

6

NS/S

7

NS

⋮

⋮

⋮

⋮

⋮

⋮

no S only  
handlers  
lower than  
highest  
NS handler

Fig. 34

30/62

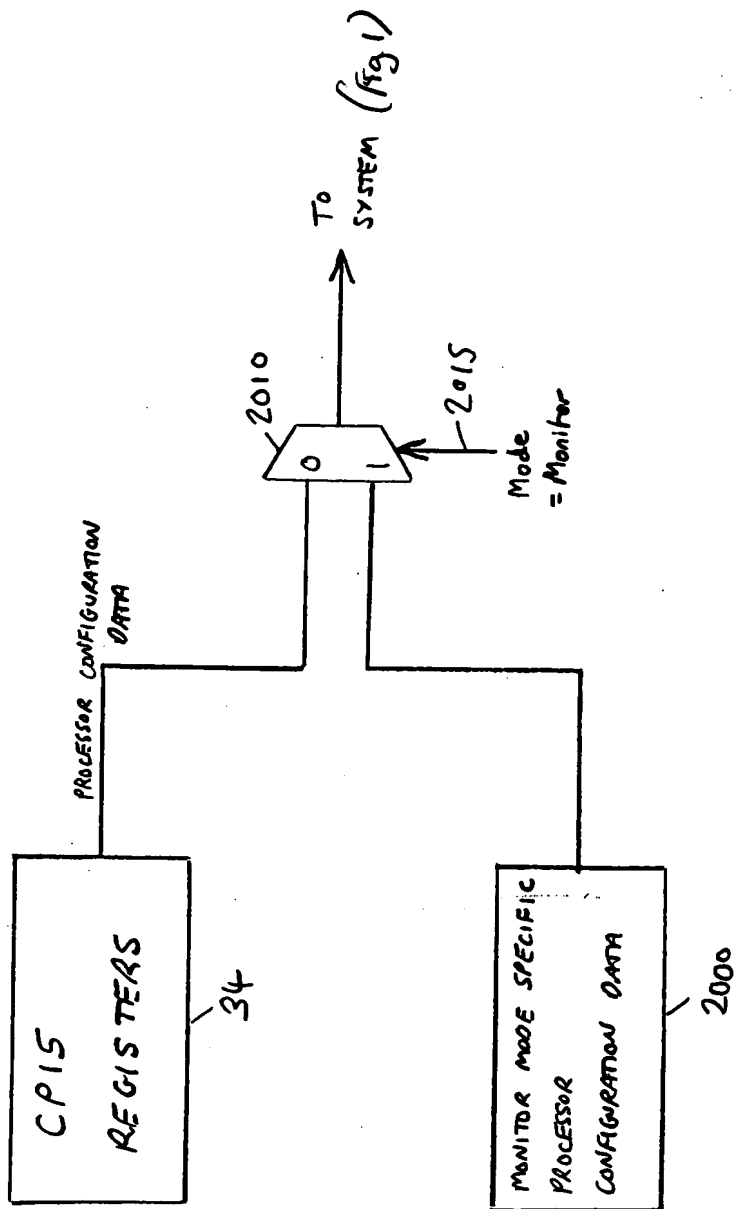


FIG 35

31/62

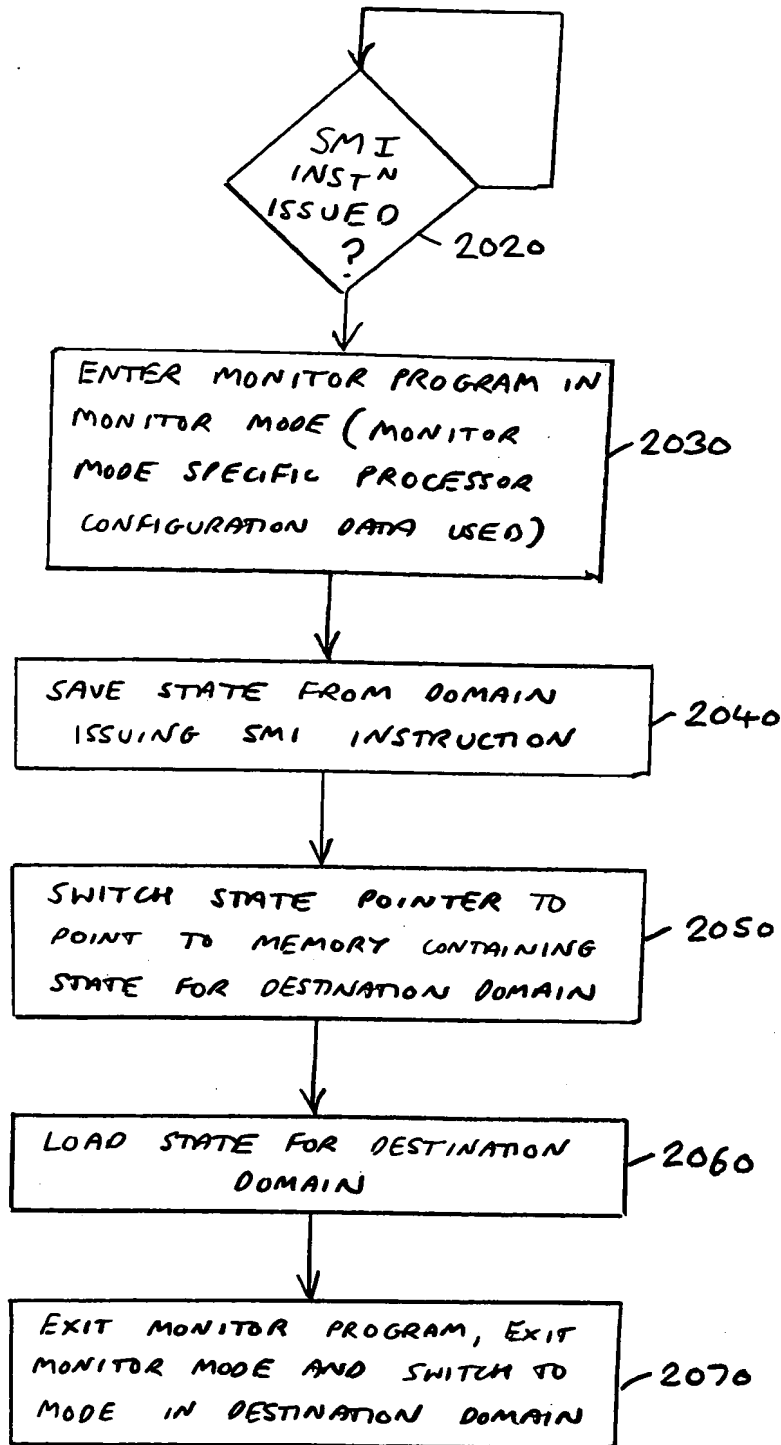


FIG 36



32/62

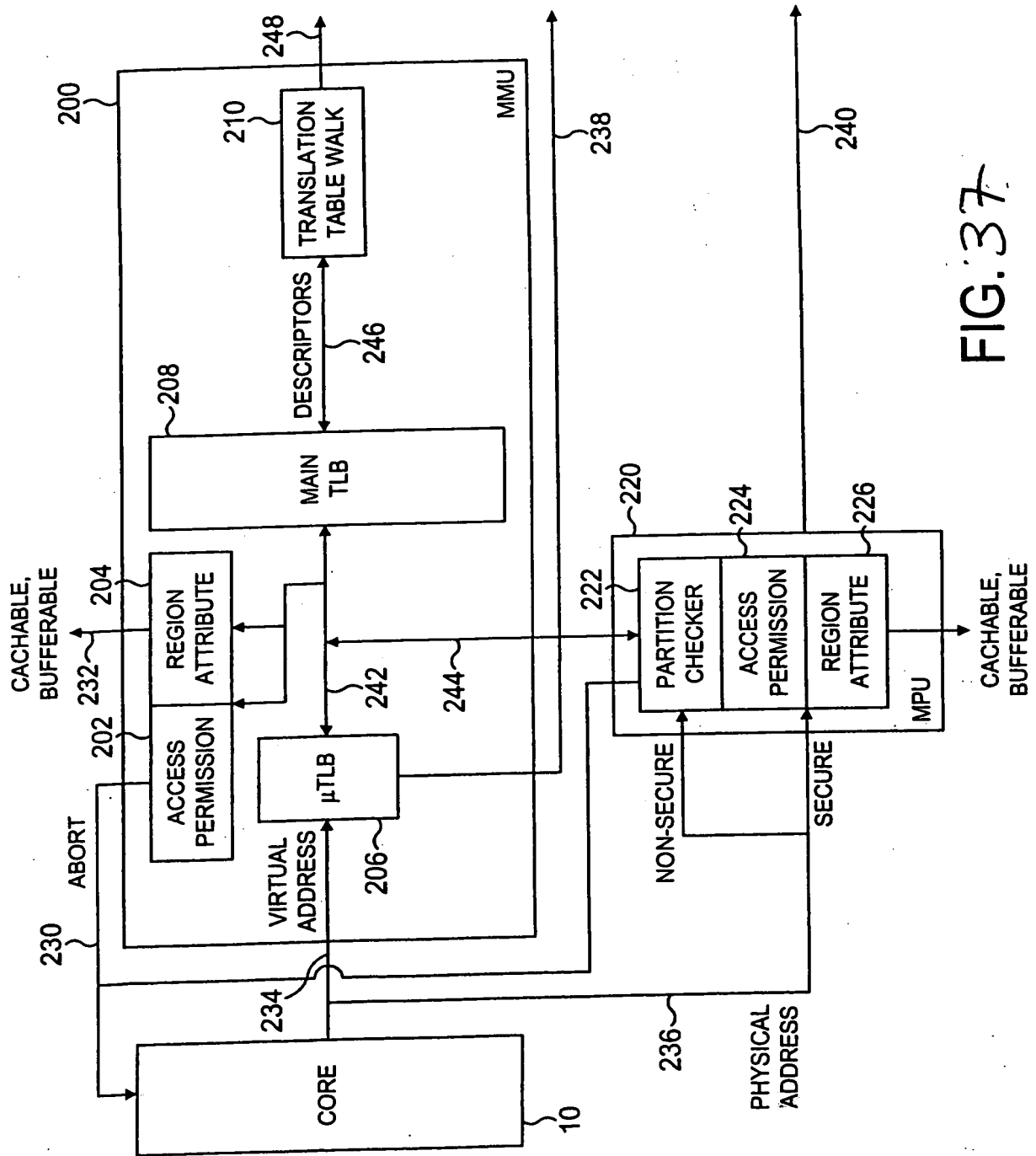


FIG. 37

FIG. 38 is a block diagram of MMU 200. The MMU 200 is connected to a CORE 10. The MMU 200 contains several internal components: a CACHABLE, BUFFERABLE block (230) which receives an ABORT signal and sends data to the CORE 10; an ACCESS PERMISSION block (202) which receives data from the CORE 10 and sends it to the REGION ATTRIBUTE block (204); a REGION ATTRIBUTE block (204) which sends data to the MAIN TLB (208); a μTLB block (234) which receives data from the CORE 10 and sends it to the MAIN TLB (208); a PARTITION CHECKER block (222) which receives data from the CORE 10 and sends it to the MAIN TLB (208); a TRANSLATION TABLE WALK block (210) which receives data from the MAIN TLB (208) and sends it to the CORE 10; and a DEScriptors block (246) which receives data from the MAIN TLB (208) and sends it to the TRANSLATION TABLE WALK block (210). The MMU 200 is also connected to a CORE 10 via a bus (238).

FIG. 3b

34/62

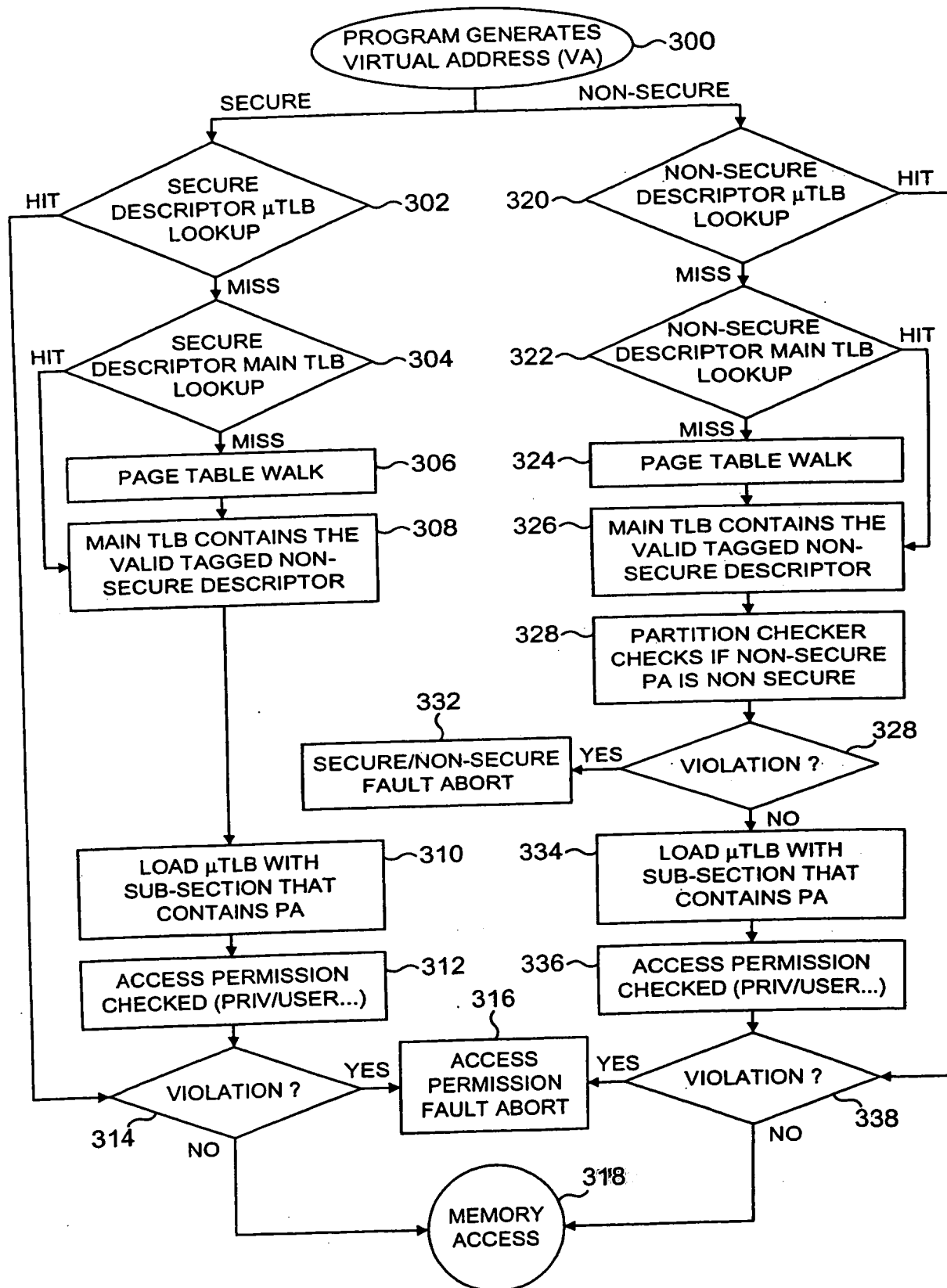


FIG. 39

35/62

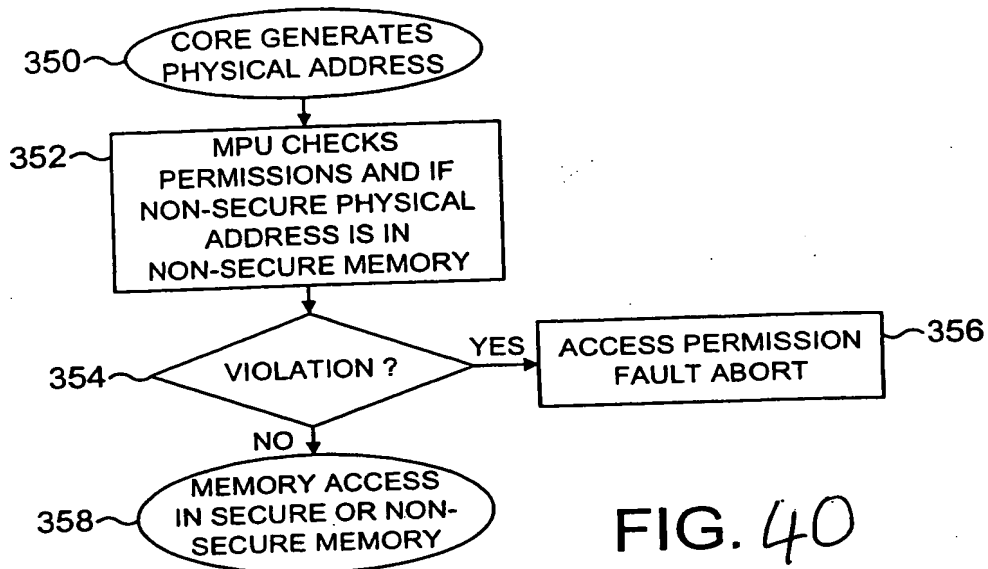


FIG. 40

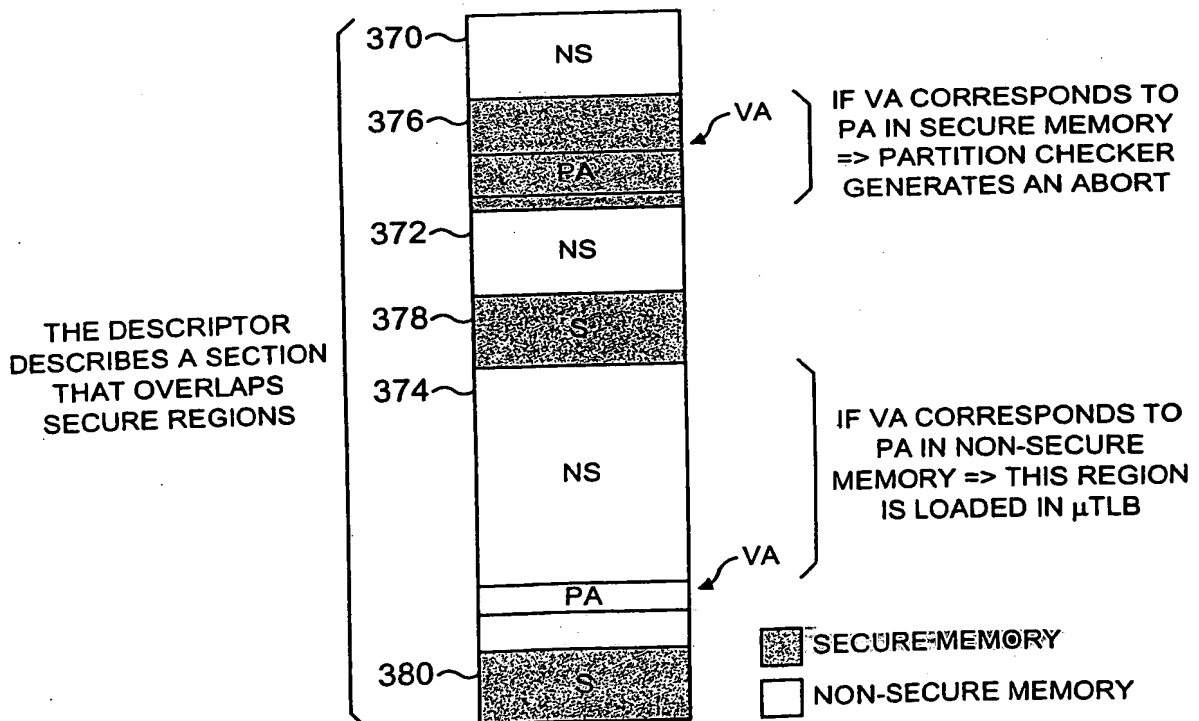


FIG. 41

36/62

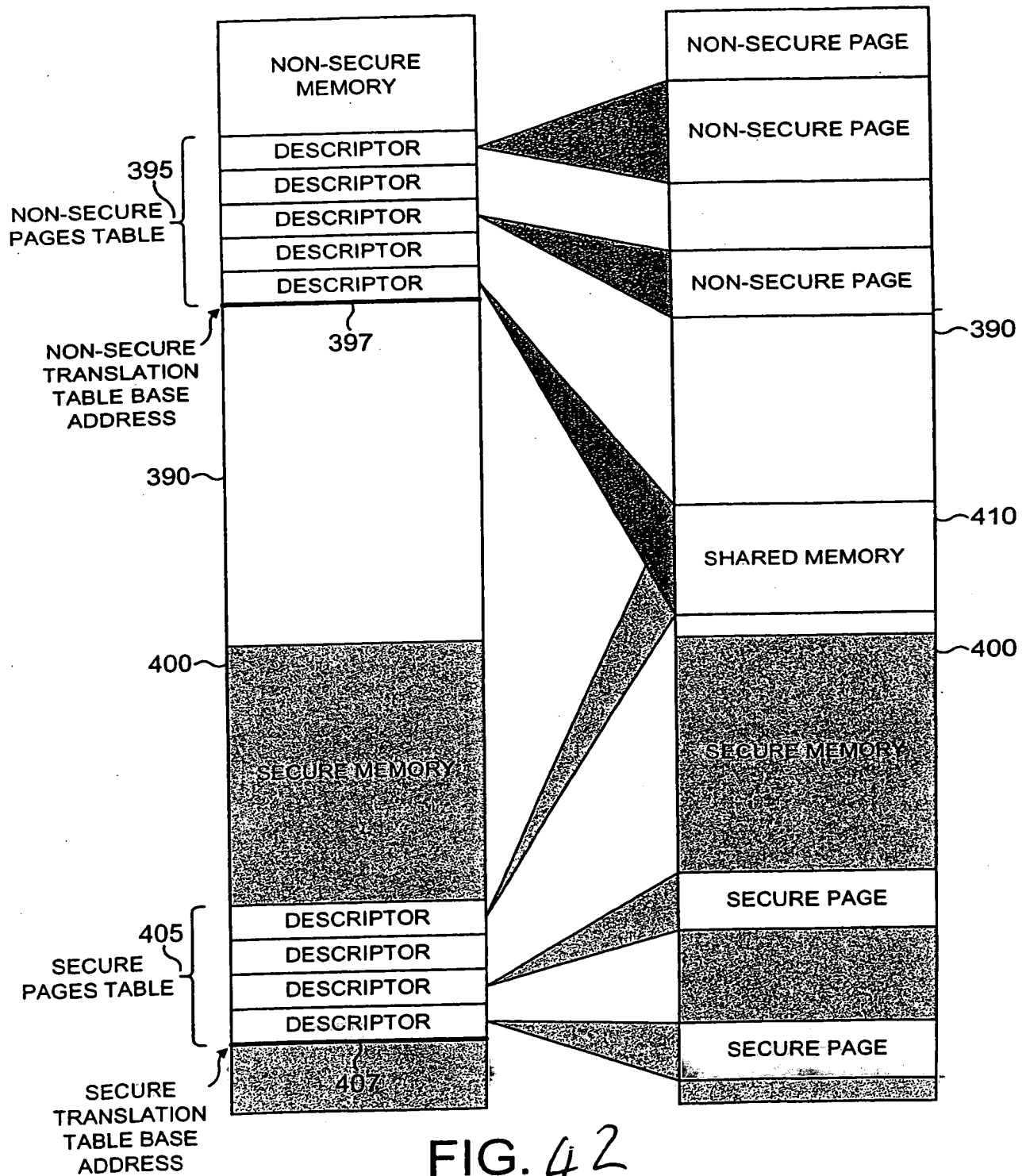


FIG. 42

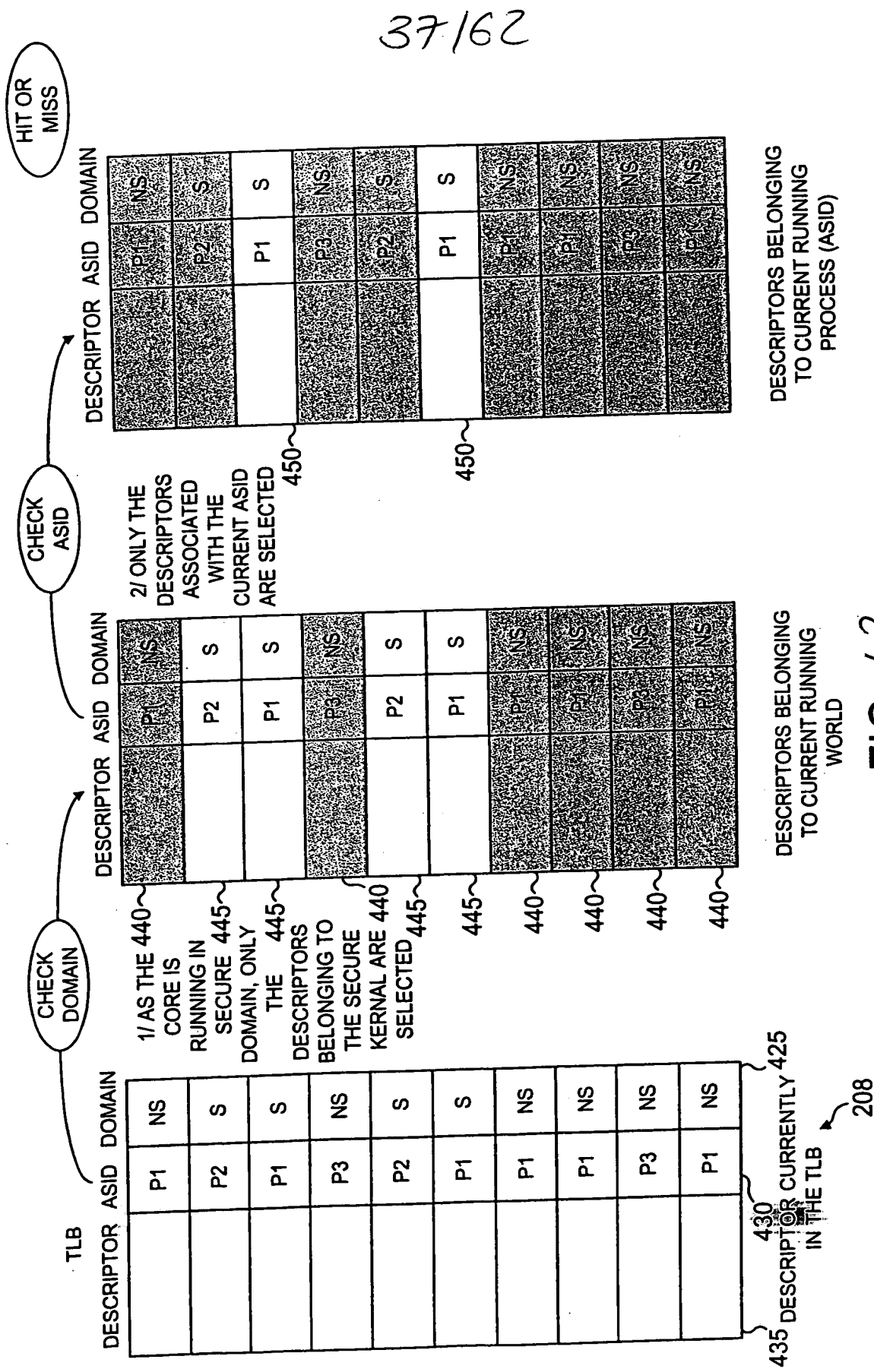


FIG. 43

38/62

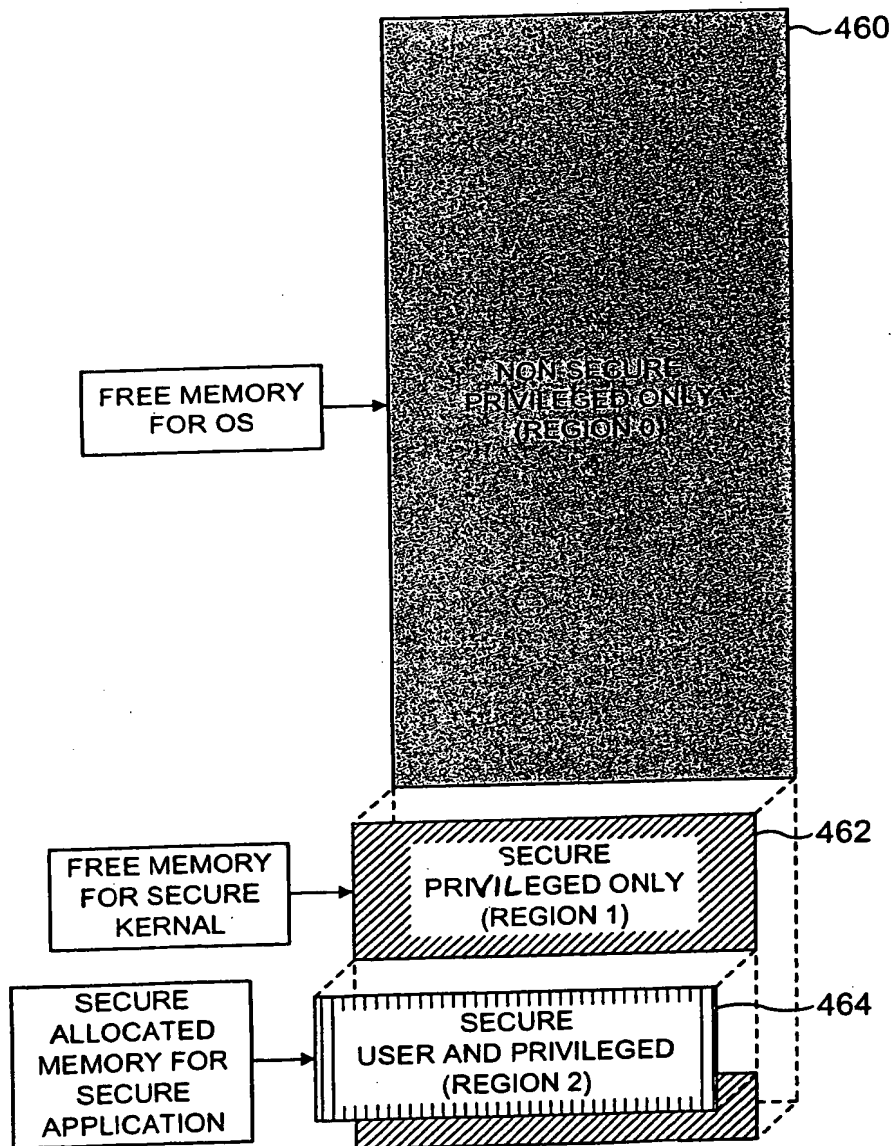


FIG. 44

39/62

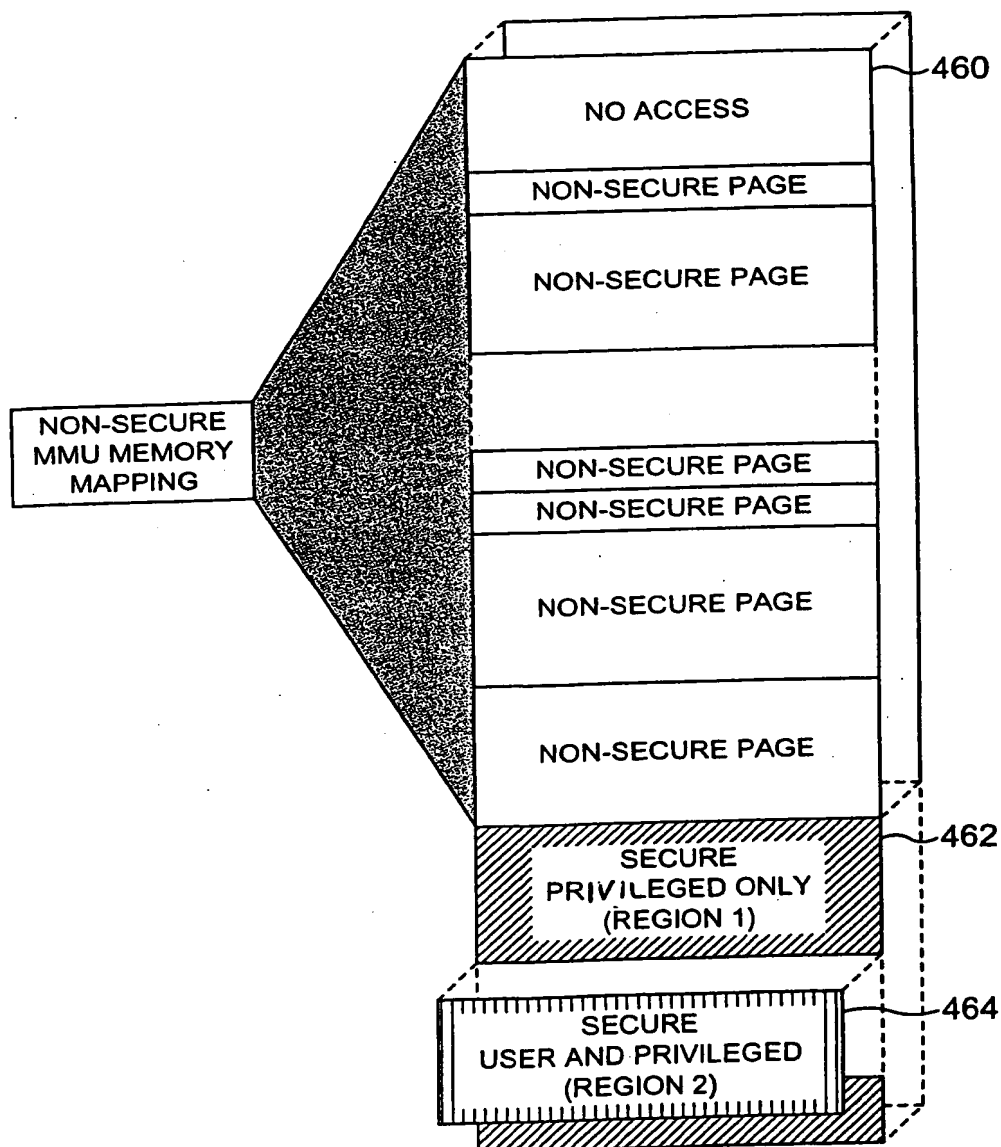


FIG. 45



40/62

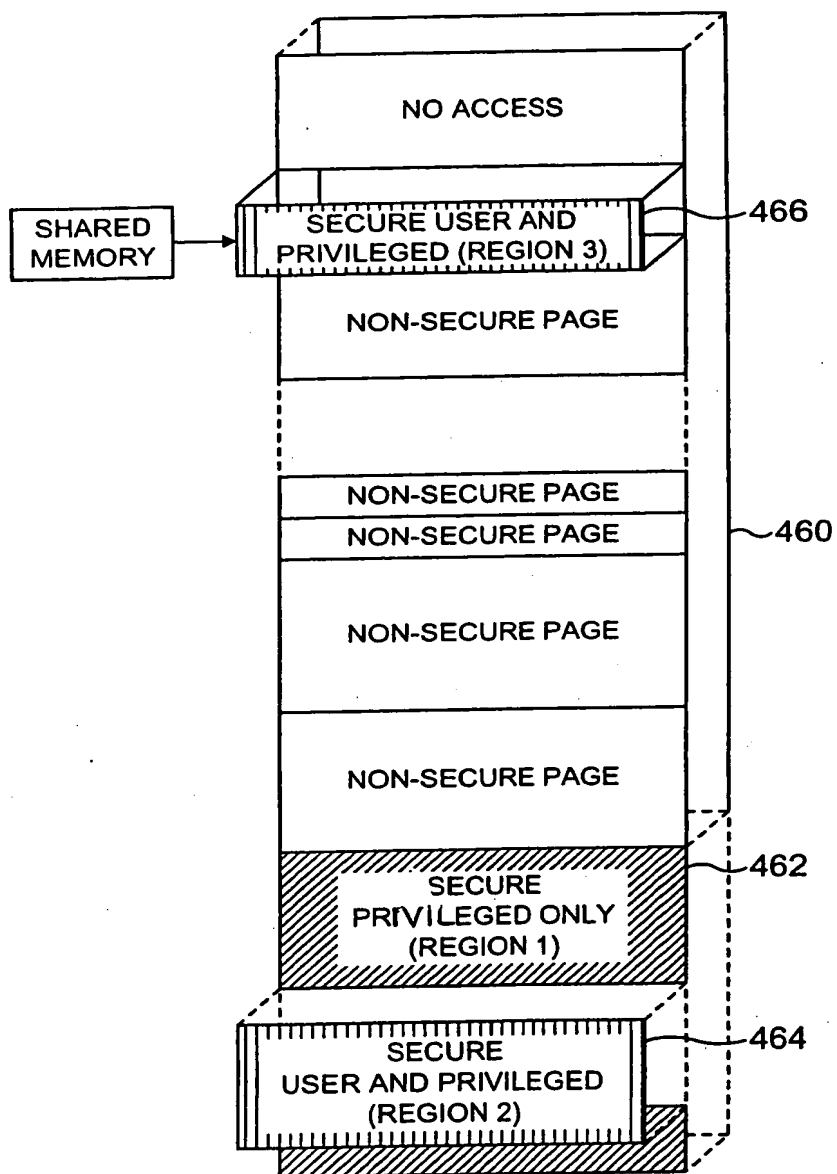


FIG. 46

41/62

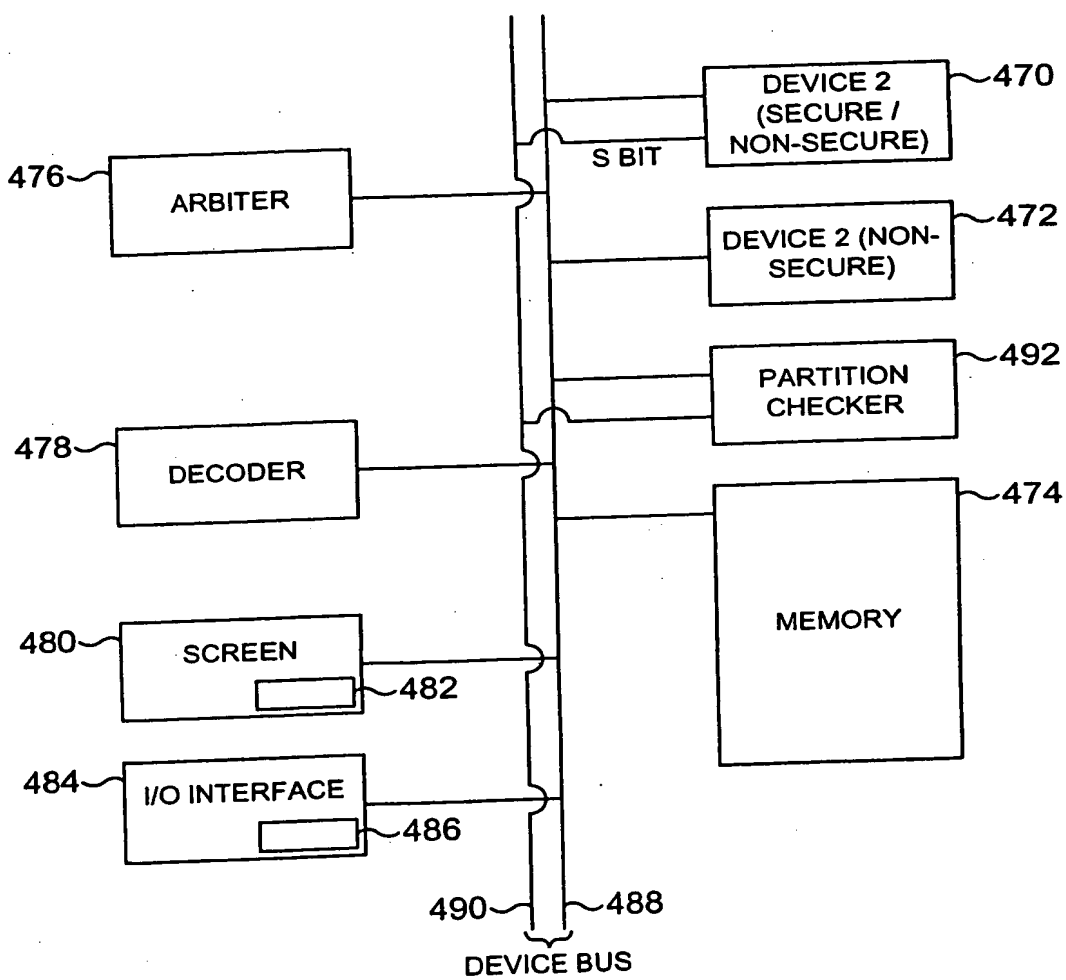


FIG. 47

42/62

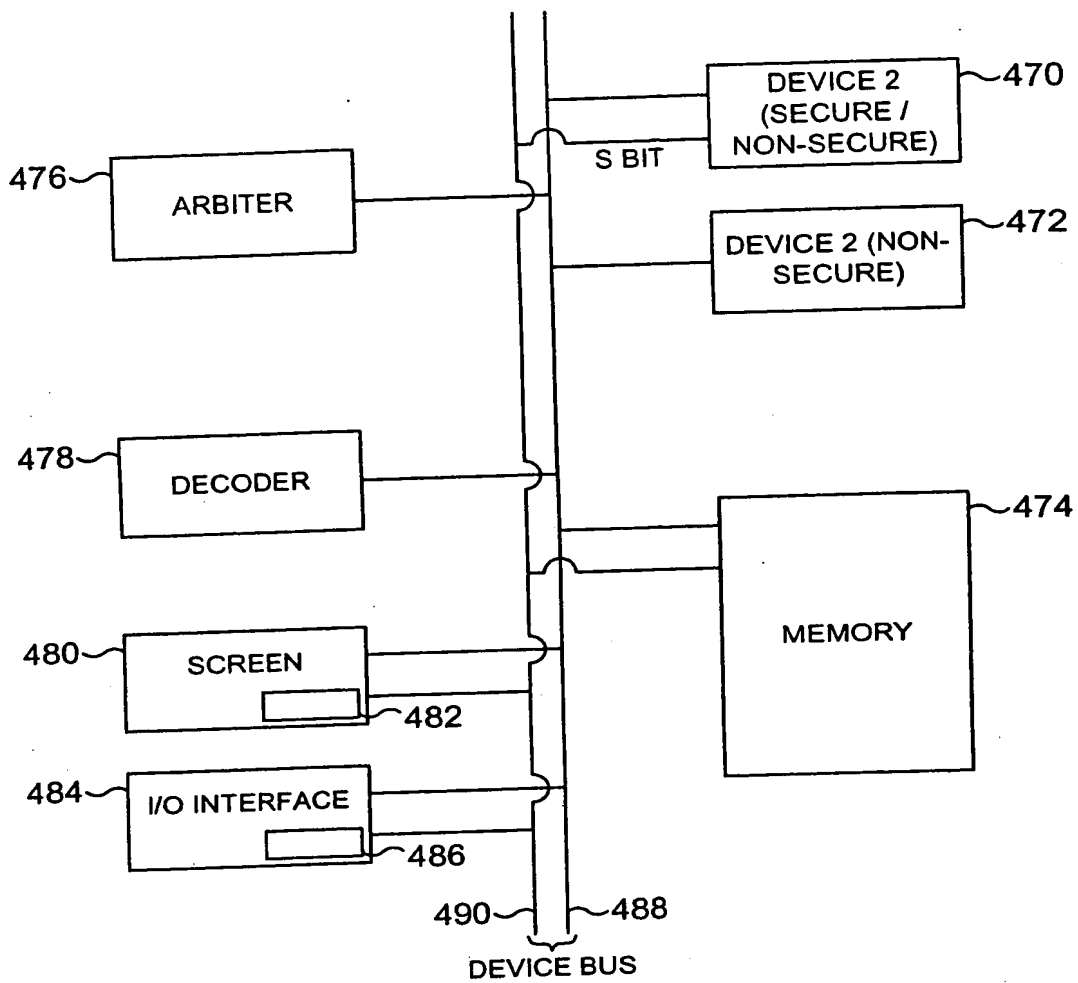


FIG. 48

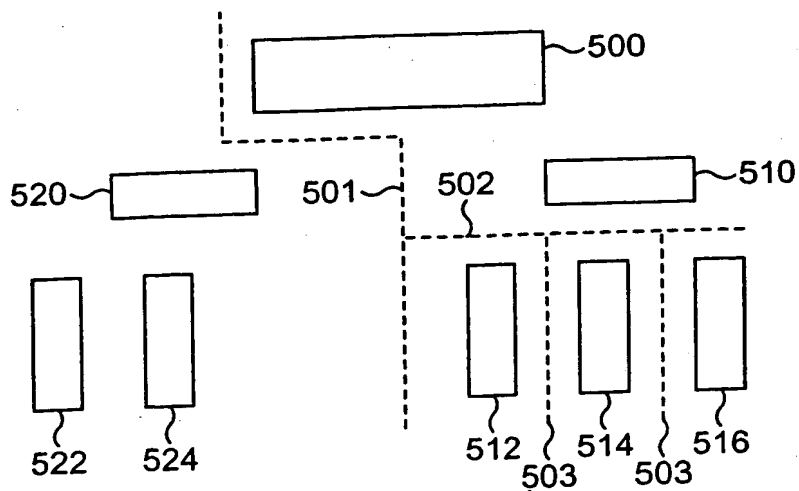
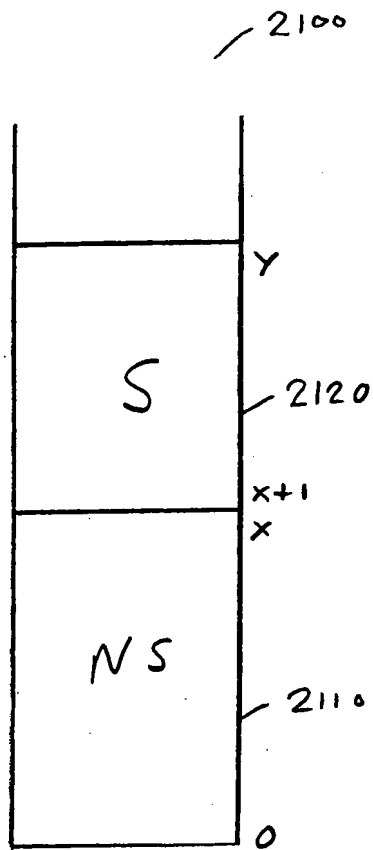


FIG. 59

43/62



PHYSICAL  
ADDRESS SPACE

FIG. 49

44/62

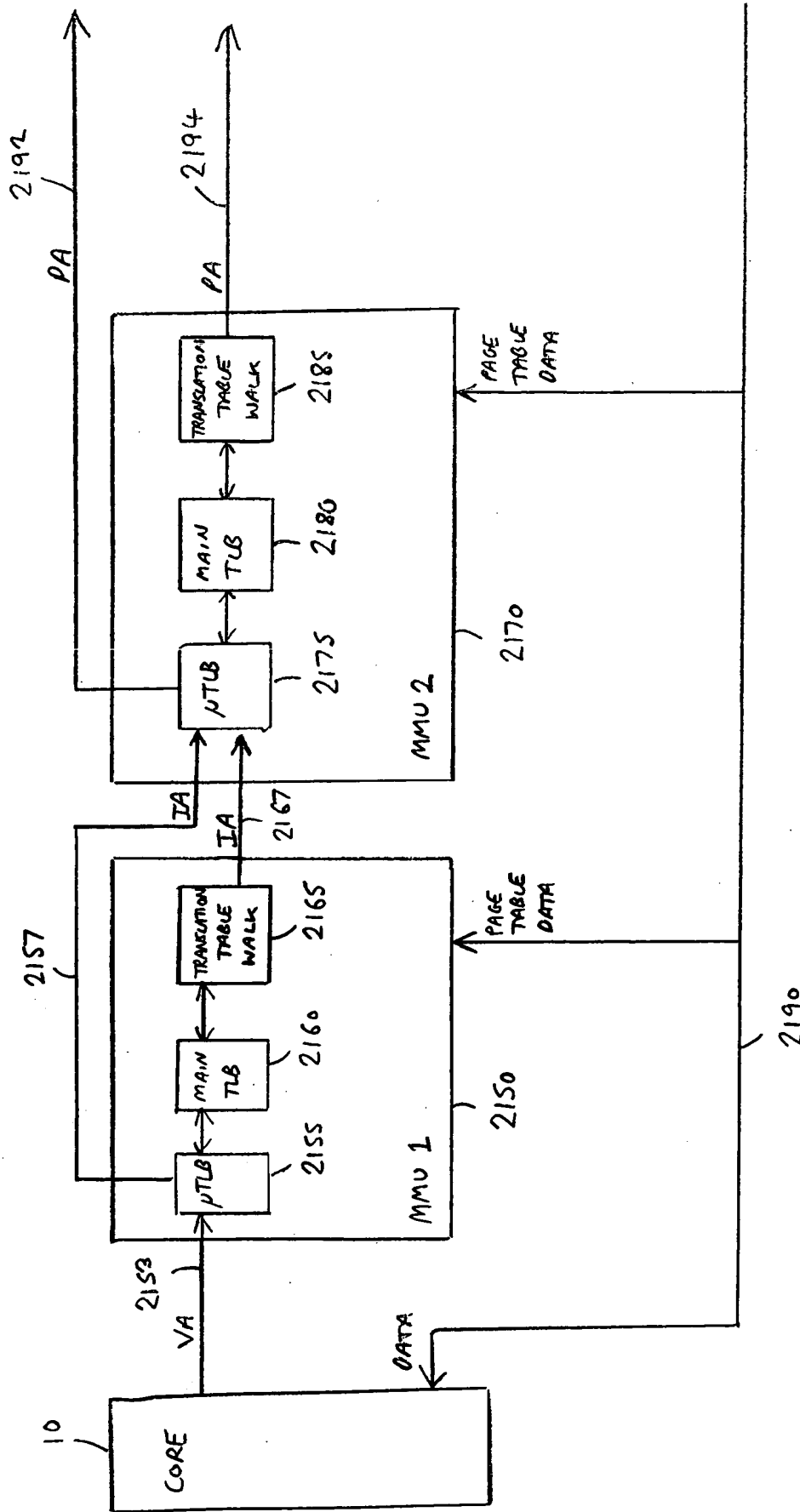


FIG 50A

45/62

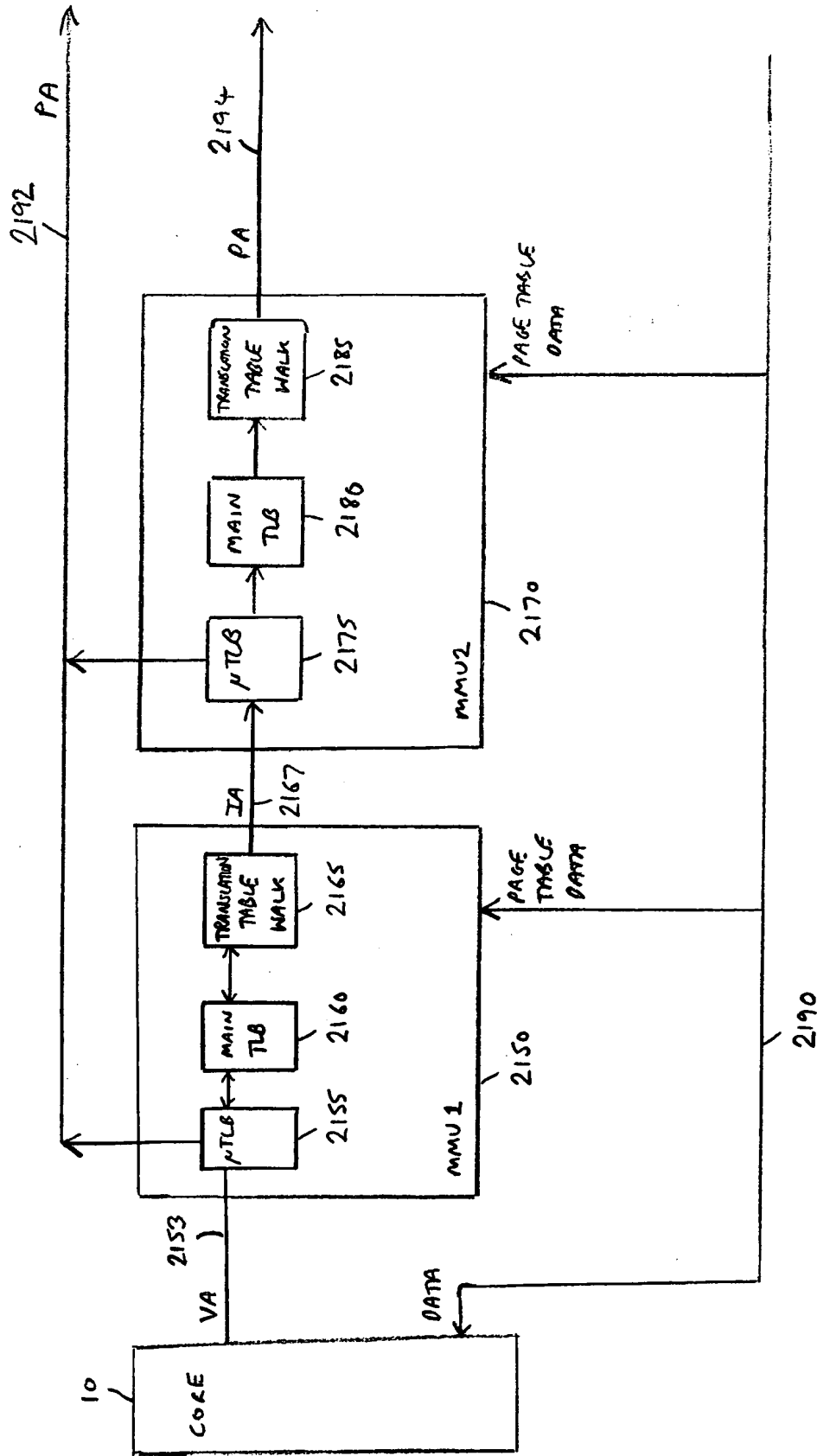


FIG 50B

46/62

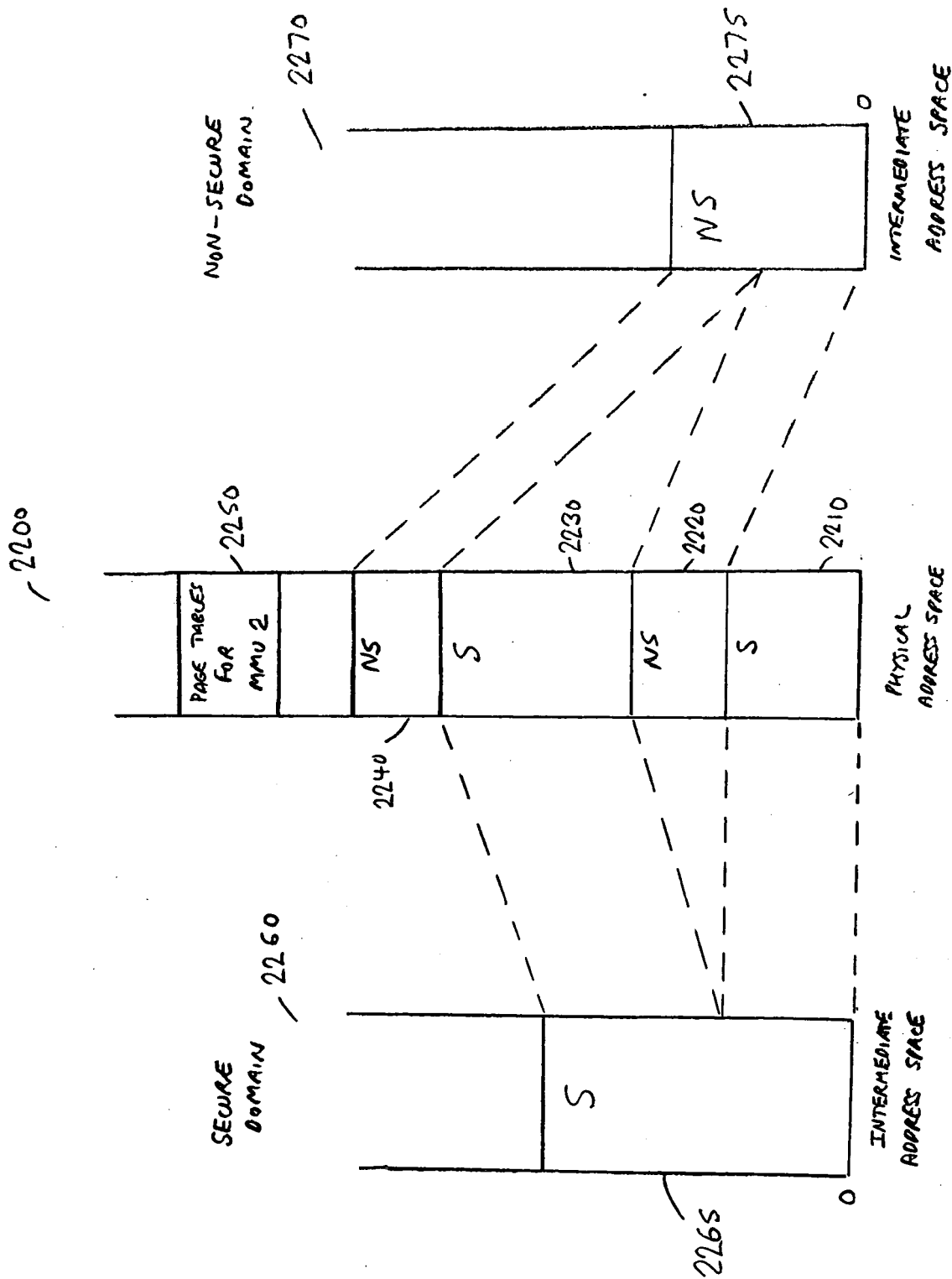


FIG 51

47/62

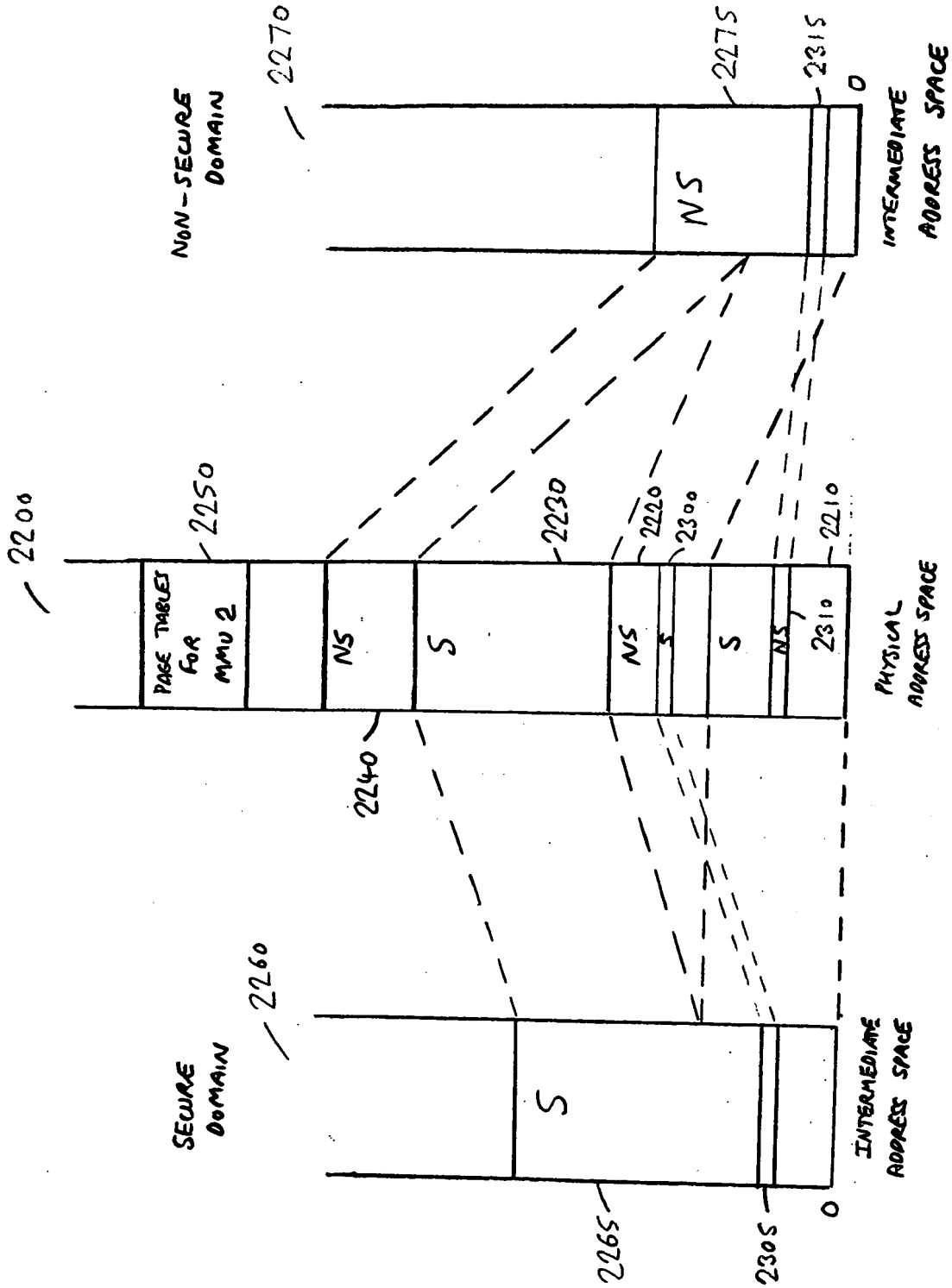


FIG 52



48/62

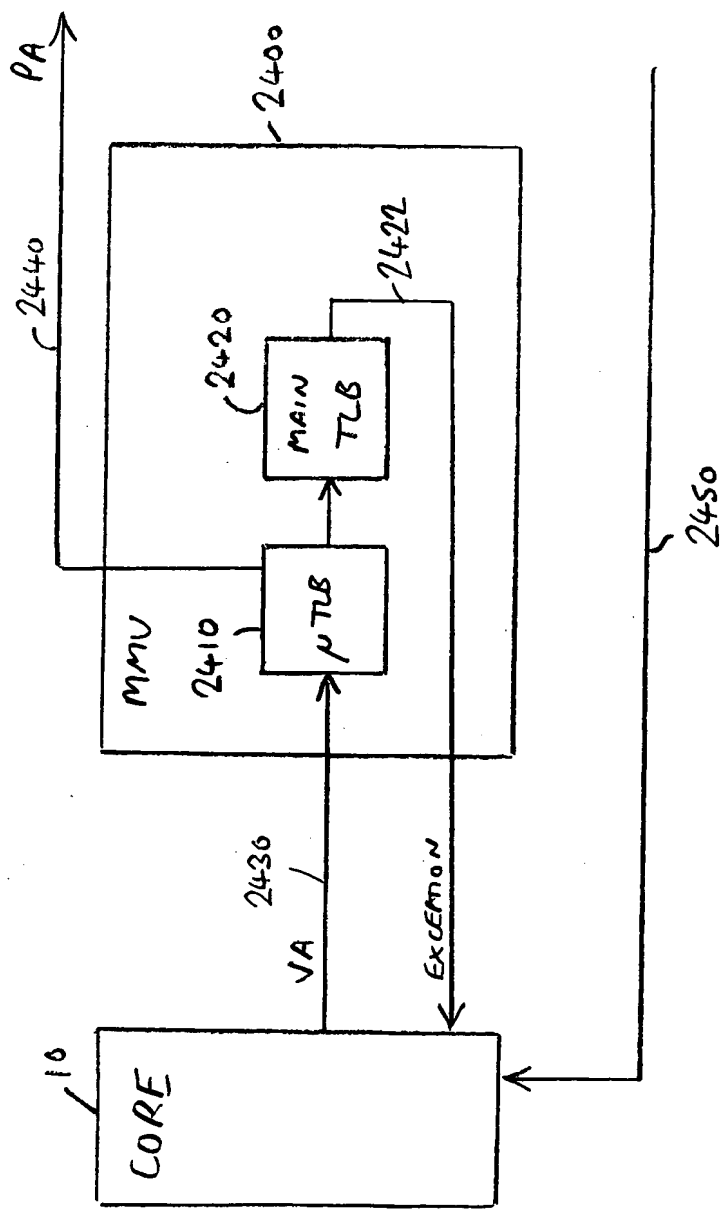


FIG 53

49/62

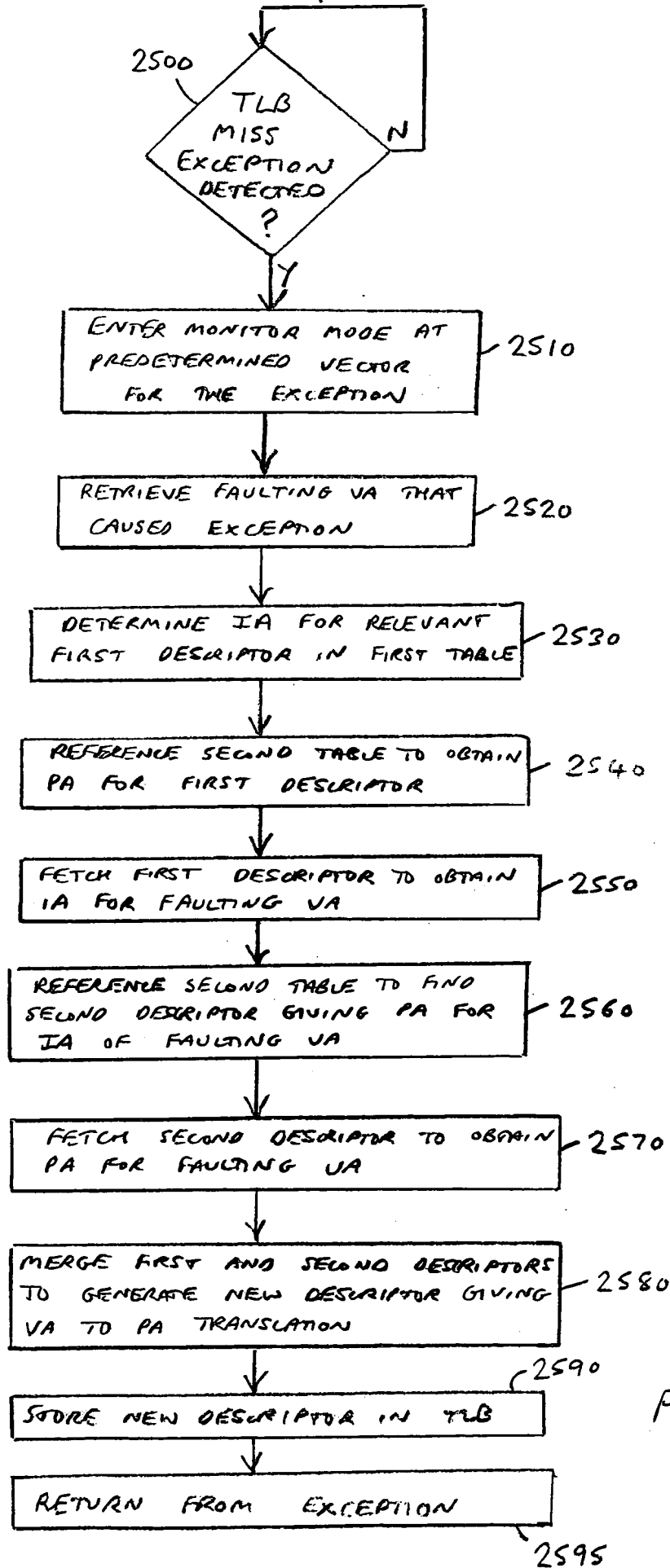


FIG. 54

50/62

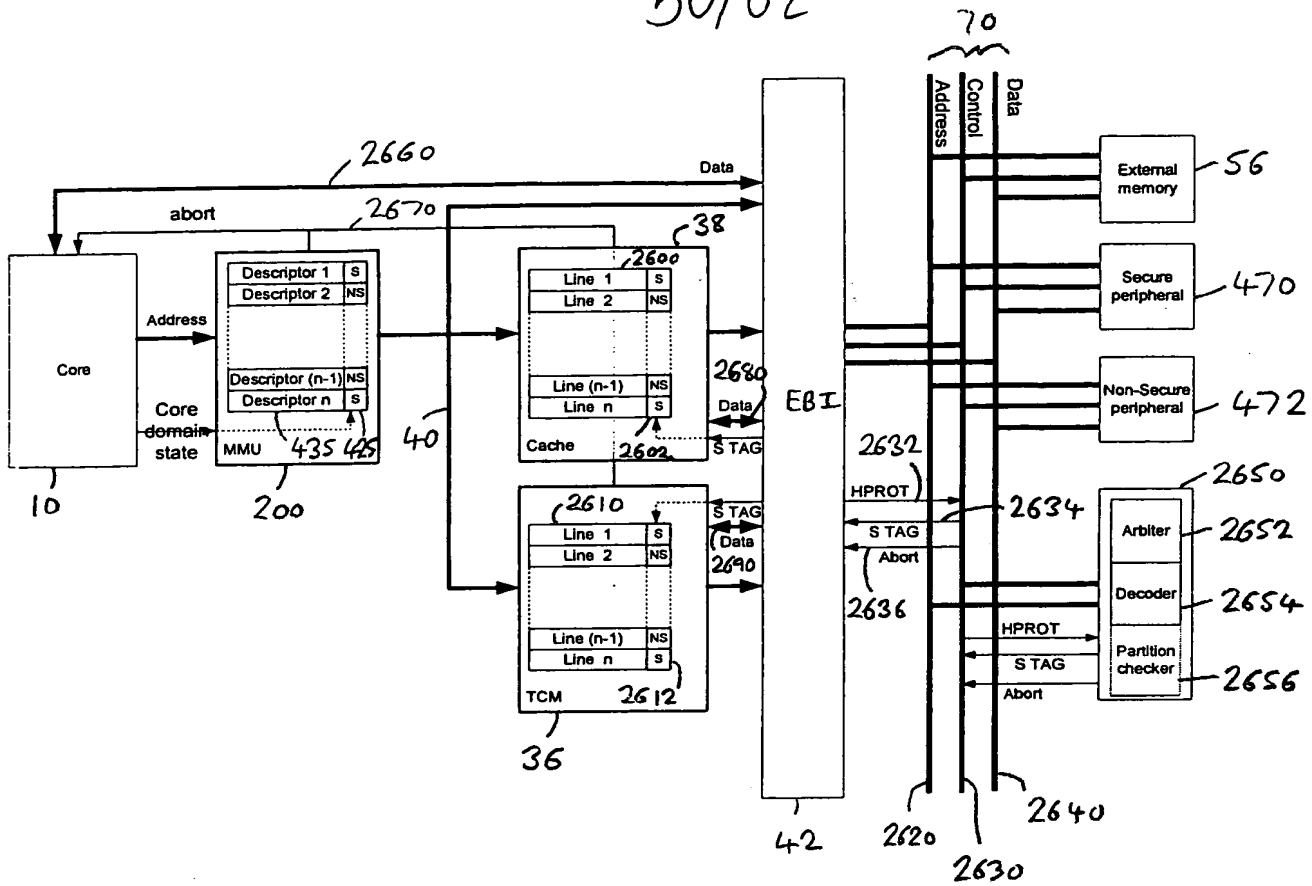


FIG 55

51/62

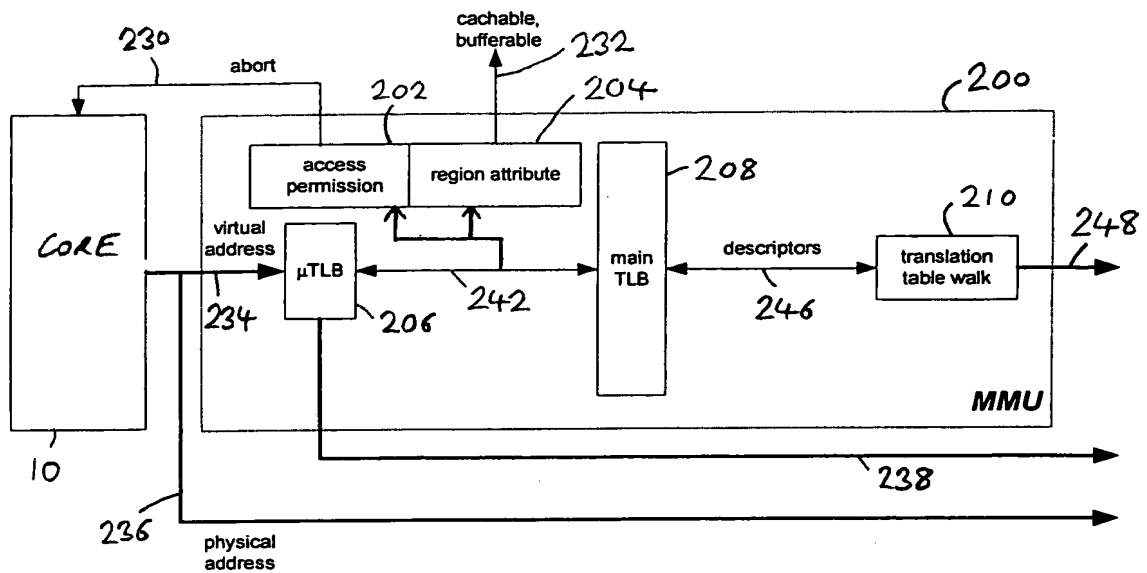


FIG 56

52/62

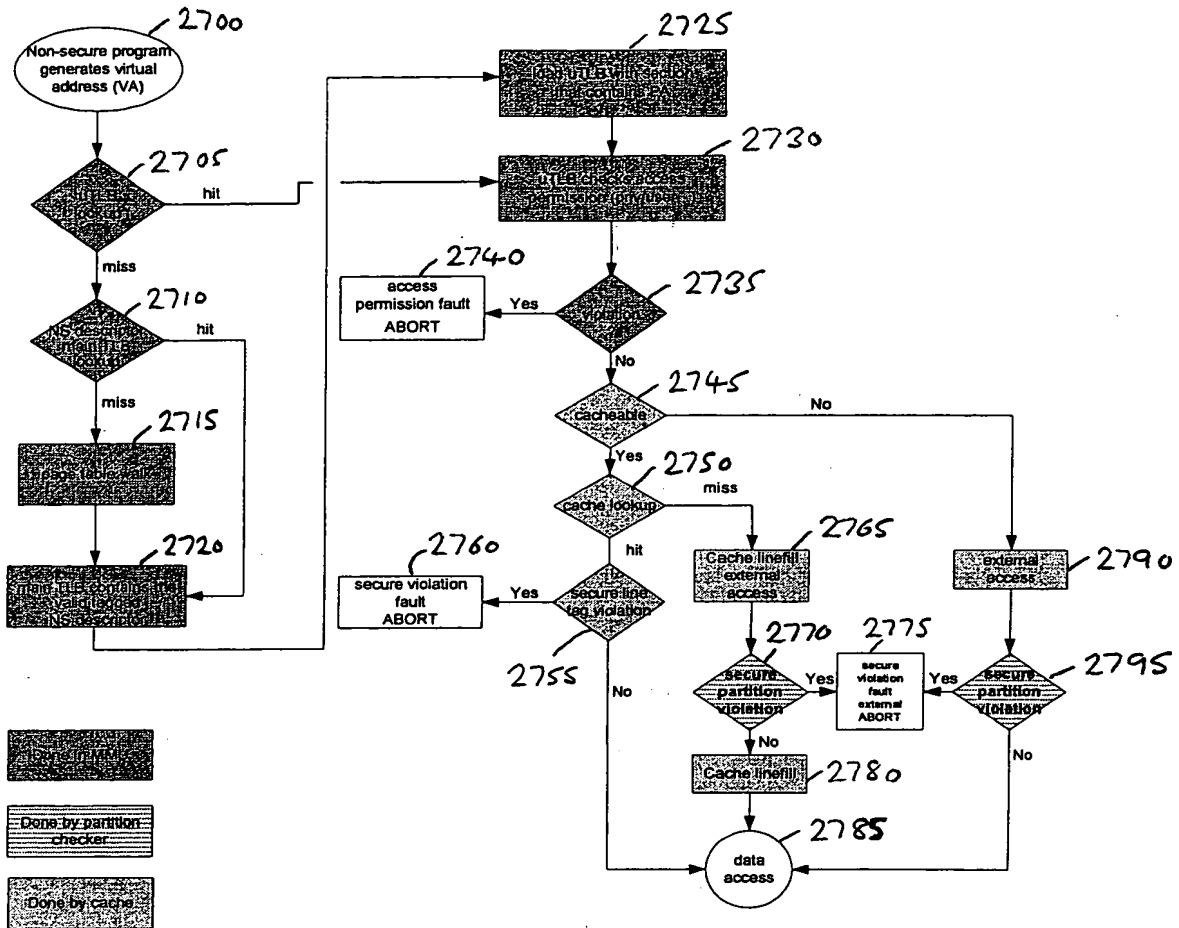


FIG 57

53/62

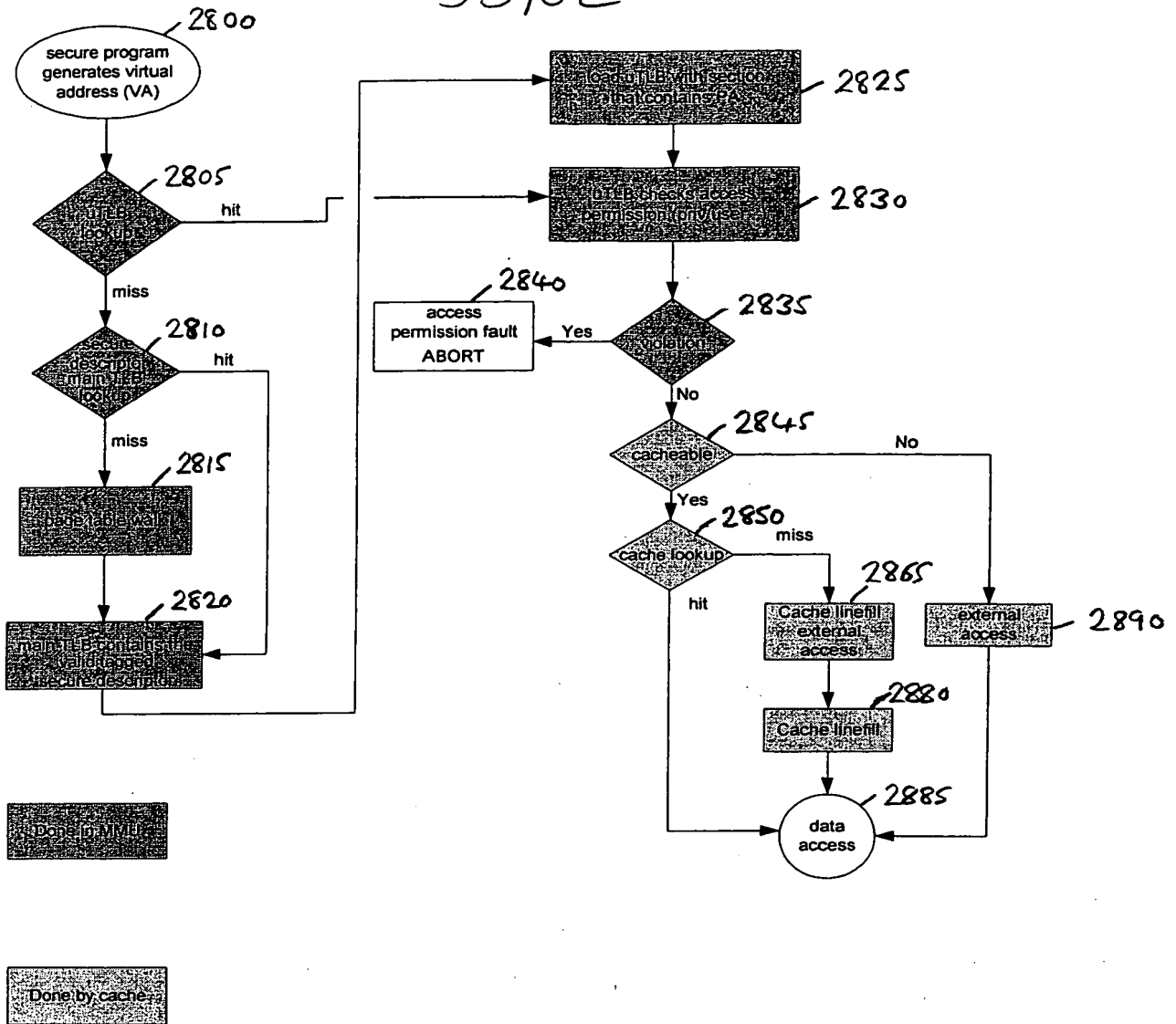


FIG 58

54/62

METHOD OF ENTRY	HOW TO PROGRAM?	HOW TO ENTER?	ENTRY MODE
BREAKPOINT HITS	DEBUG TAP OR SOFTWARE (CP14)	PROGRAM BREAKPOINT REGISTER AND/OR CONTEXT-ID AND COMPARISONS SUCCEED WITH INSTRUCTION ADDRESS AND/OR CP15 CONTEXT ID <sup>(2)</sup>	HALT/MONITOR <sup>(1)</sup>
SOFTWARE BREAKPOINT INSTRUCTION	PUT A BKPT INSTRUCTION INTO SCAN CHAIN 4 (INSTRUCTION TRANSFER REGISTER) THROUGH DEBUG TAP OR USE BKPT INSTRUCTION DIRECTLY IN THE CODE	BKPT INSTRUCTION MUST REACH EXECUTION STAGE	HALT/MONITOR
VECTOR TRAP BREAKPOINT	DEBUG TAP	PROGRAM VECTOR TRAP REGISTER AND ADDRESS MATCHES	HALT/MONITOR
WATCHPOINT HITS	DEBUG TAP OR SOFTWARE (CP14)	PROGRAM WATCHPOINT REGISTER AND/OR CONTEXT-ID REGISTER AND COMPARISONS SUCCEED WITH INSTRUCTION ADDRESS AND/OR CP15 CONTEXT ID <sup>(2)</sup>	HALT/MONITOR <sup>(1)</sup>
INTERNAL DEBUG REQUEST	DEBUG TAP	HALT INSTRUCTION HAS BEEN SCANNED IN	HALT
EXTERNAL DEBUG REQUEST	NOT APPLICABLE	EDBGRQ INPUT PIN IS ASSERTED	HALT

<sup>(1)</sup>: IN MONITOR, BREAKPOINTS AND WATCHPOINTS CANNOT BE DATA-DEPENDENT.

<sup>(2)</sup>: THE CORES HAVE SUPPORT FOR THREAD-AWARE BREAKPOINTS AND WATCHPOINTS IN ORDER TO ABLE TO ENABLE SECURE DEBUG ON SOME PARTICULAR THREADS.

FIG. 60

55/62




NAME	MEANING	RESET VALUE	ACCESS	INSERTED IN SCAN CHAIN FOR TEST
MONITOR MODE ENABLE BIT	0: HALT MODE 1: MONITOR MODE	1	R/W BY PROGRAMMING THE ICE BY THE JTAG (SCAN 1)  •R/W BY USING MRC/MCR INSTRUCTION (CP14)	YES
SECURE DEBUG ENABLE BIT	0: DEBUG IN NON-SECURE WORLD ONLY 1: DEBUG IN SECURE WORLD AND NON-SECURE WORLD	0	IN FUNCTIONAL MODE OR DEBUG MONITOR MODE: R/W BY USING MRC/MCR INSTRUCTION (CP14) (ONLY IN SECURE SUPERVISOR MODE) IN DEBUG HALT MODE: NO ACCESS - MCR/MRC INSTRUCTIONS HAVE ANY EFFECT  (R/W BY PROGRAMMING THE ICE BY THE JTAG (SCAN 1) IF JSDAEN=1)	NO
SECURE TRACE ENABLE BIT	0: ETM IS ENABLED IN NON-SECURE WORLD ONLY. 1: ETM IS ENABLED IN SECURE WORLD AND NON-SECURE WORLD	0	IN FUNCTIONAL MODE OR DEBUG MONITOR MODE: R/W BY USING MRC/MCR INSTRUCTION (CP14) (ONLY IN SECURE SUPERVISOR MODE) IN DEBUG HALT MODE: NO ACCESS - MCR/MRC INSTRUCTIONS HAVE ANY EFFECT  (R/W BY PROGRAMMING THE ICE BY THE JTAG (SCAN 1) IF JSDAEN=1)	NO
SECURE USER-MODE ENABLE BIT	0: DEBUG IS NOT POSSIBLE IN SECURE USER MODE 1: DEBUG IS POSSIBLE IN SECURE USER MODE	1	IN FUNCTIONAL MODE OR DEBUG MONITOR MODE: R/W BY USING MRC/MCR INSTRUCTION (CP14) (ONLY IN SECURE SUPERVISOR MODE) IN DEBUG HALT MODE: NO ACCESS - MCR/MRC INSTRUCTIONS HAVE ANY EFFECT  (R/W BY PROGRAMMING THE ICE BY THE JTAG (SCAN 1) IF JSDAEN=1)	NO
SECURE THREAD-AWARE ENABLE BIT	0: DEBUG IS NOT POSSIBLE FOR A PARTICULAR THREAD 1: DEBUG IS POSSIBLE FOR A PARTICULAR THREAD	0	IN FUNCTIONAL MODE OR DEBUG MONITOR MODE: R/W BY USING MRC/MCR INSTRUCTION (CP14) (ONLY IN SECURE SUPERVISOR MODE) IN DEBUG HALT MODE: NO ACCESS - MCR/MRC INSTRUCTIONS HAVE ANY EFFECT  (R/W BY PROGRAMMING THE ICE BY THE JTAG (SCAN 1) IF JSDAEN=1)	NO

FIG. 61



56162

FUNCTION TABLE

D	CK	Q[n+1]
0		0
1		1
X		Q[n]

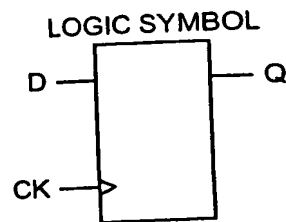







FIG. 62

FUNCTION TABLE

D	SI	SE	CK	Q[n+1]
0	X	0		0
1	X	0		1
X	X	X		Q[n]
X	0	1		0
X	1	1		1

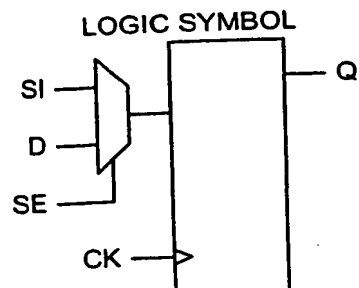


FIG. 63

57/62

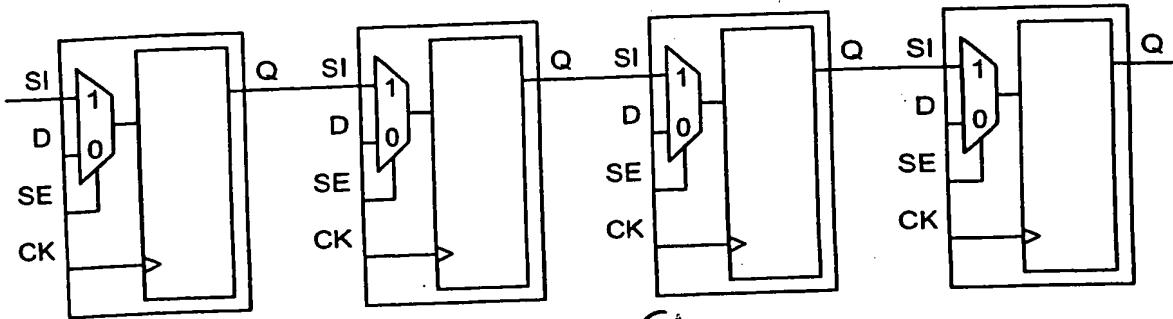


FIG. 64

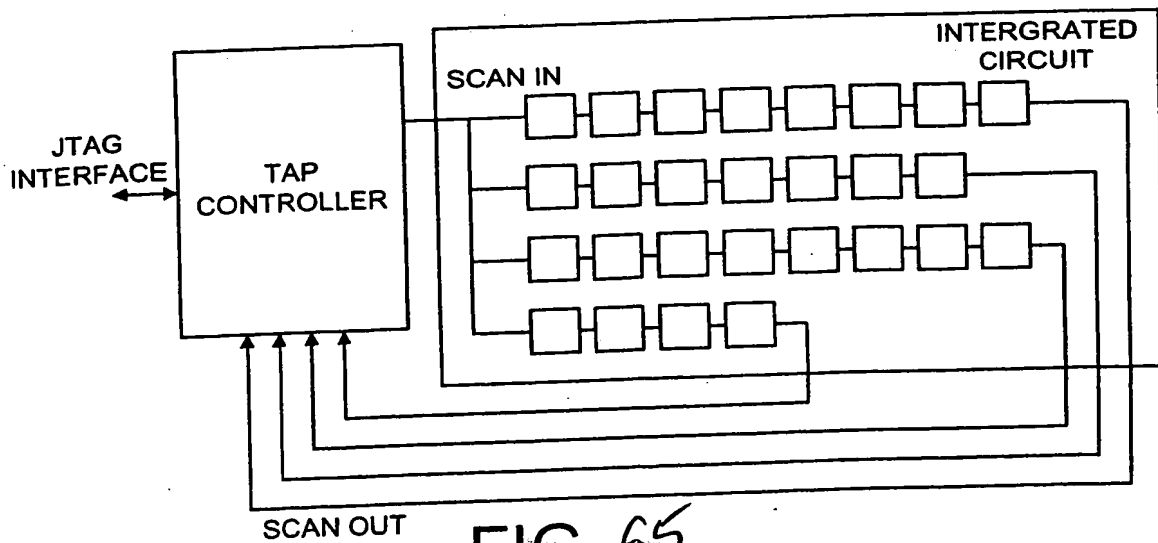


FIG. 65

58/62

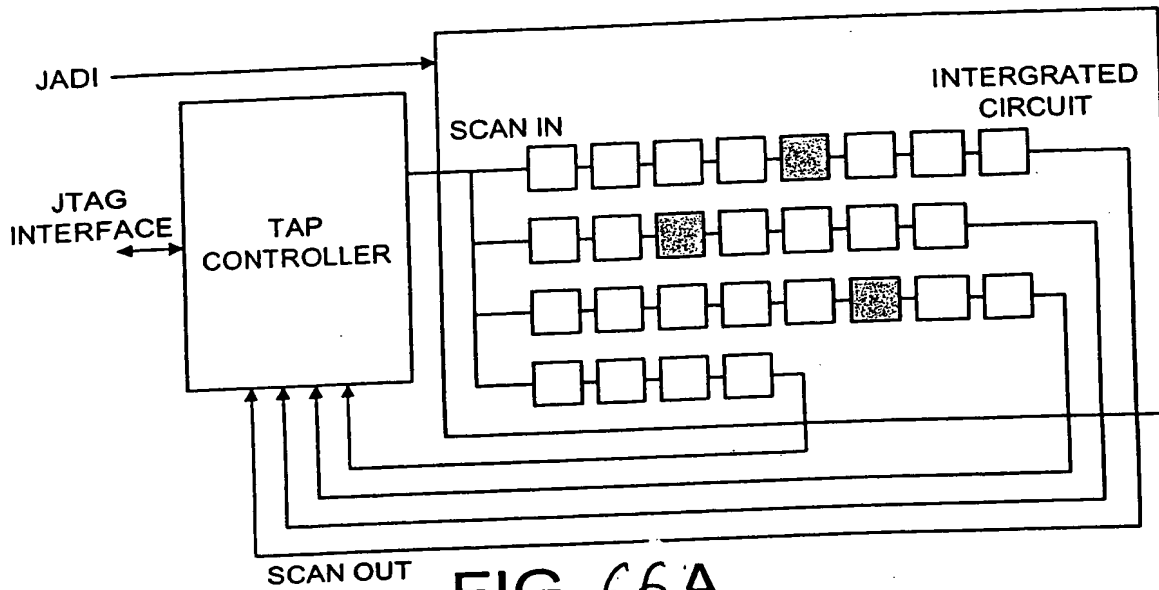


FIG. 66A

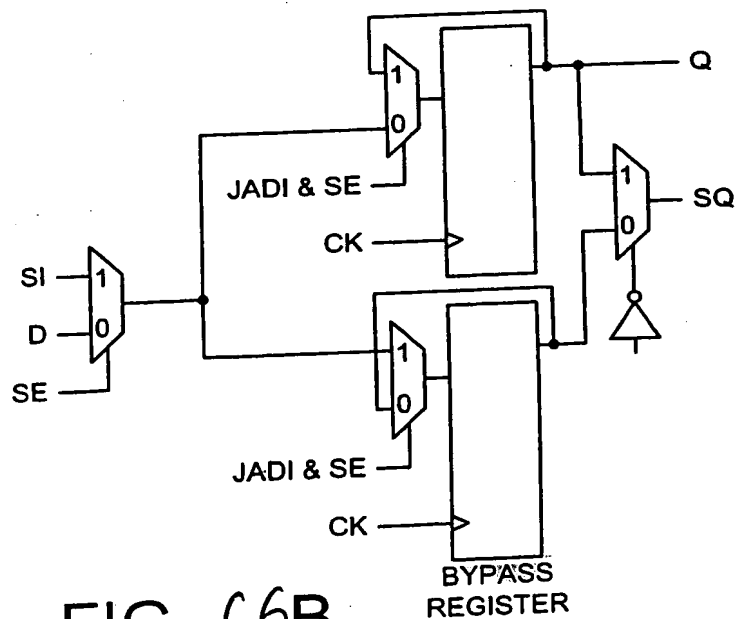


FIG. 66B

59/62

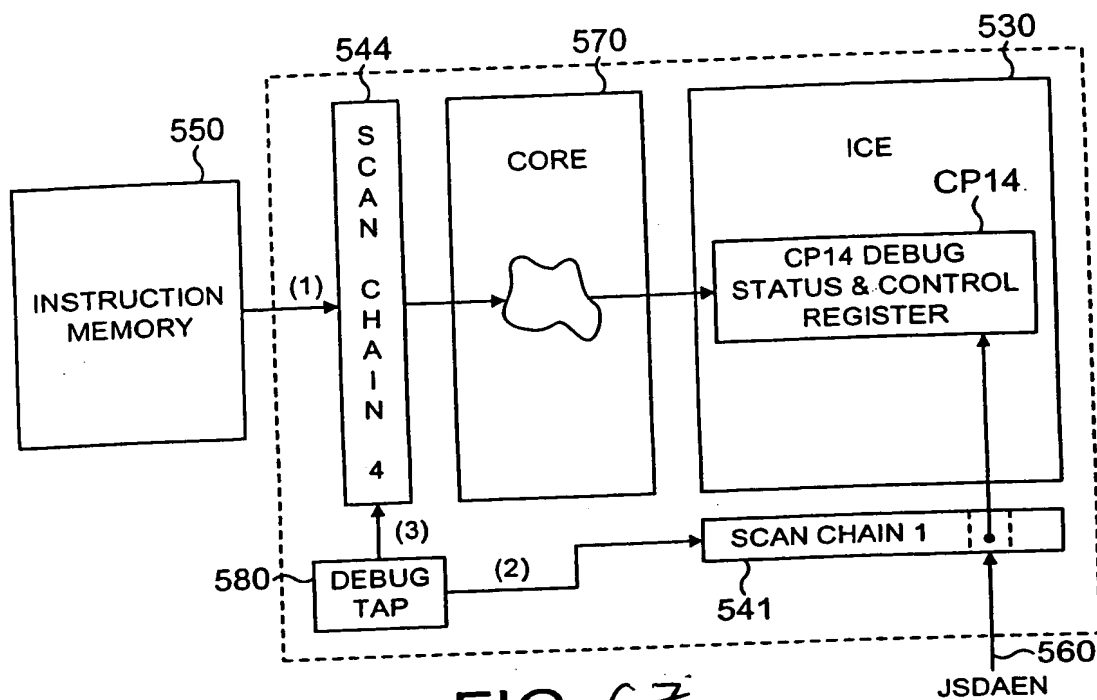


FIG. 67

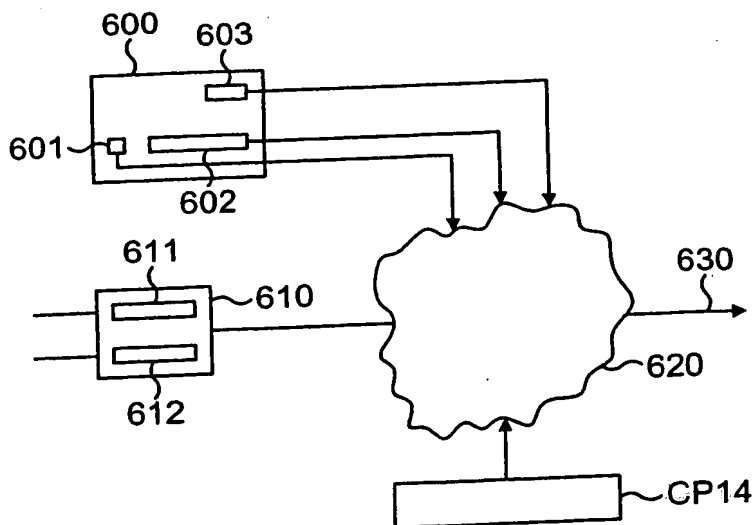


FIG. 68

60/62

CP14 BITS IN DEBUG AND STATUS CONTROL REGISTER				MEANING
SECURE DEBUG ENABLE BIT	SECURE USER- MODE DEBUG ENABLE BIT	SECURE THREAD- AWARE DEBUG ENABLE BIT		
0	X	X		NO INTRUSIVE DEBUG IN ENTIRE WORLD IS POSSIBLE. ANY DEBUG REQUEST, BREAKPOINTS, WATCHPOINTS, AND OTHER MECHANISM TO ENTER DEBUG STATE ARE IGNORED IN ENTIRE SECURE WORLD
1	0	X		DEBUG IN ENTIRE SECURE WORLD IS POSSIBLE
1				DEBUG IN SECURE USER-MODE ONLY. ANY DEBUG REQUEST, BREAKPOINTS, WATCHPOINTS, AND OTHER MECHANISM TO ENTER DEBUG STATE ARE TAKEN INTO ACCOUNT IN USER MODE ONLY. (BREAKPOINTS AND WATCHPOINTS LINKED OR NOT TO A THREAD ID ARE TAKEN INTO ACCOUNT). ACCESS IN DEBUG IS RESTRICTED TO WHAT SECURE USER CAN HAVE ACCESS TO.
1		X		DEBUG IS POSSIBLE ONLY IN SOME PARTICULAR THREADS. IN THAT CASE ONLY THREAD-AWARE BREAKPOINTS AND WATCHPOINTS LINKED TO A THREAD ID ARE TAKEN INTO ACCOUNT TO ENTER DEBUG STATE. EACH THREAD CAN MOREOVER DEBUG ITS OWN CODE, AND ONLY ITS OWN CODE.

FIG. 69A

61/62

CP14 BITS IN DEBUG AND STATUS CONTROL REGISTER			MEANING
SECURE TRACE ENABLE BIT	SECURE USER-MODE DEBUG ENABLE BIT	SECURE THREAD-AWARE DEBUG ENABLE BIT	
0	X	X	NO OBSERVATION DEBUG IN ENTIRE SECURE WORLD IS POSSIBLE. TRACE MODULE (ETM) MUST NOT TRACE INTERNAL CORE ACTIVITY
1	0	X	TRACE IN ENTIRE SECURE WORLD IS POSSIBLE
1	1	0	TRACE IS POSSIBLE WHEN THE CORE IS IN SECURE USER-MODE ONLY
1	1	1	TRACE IS POSSIBLE ONLY WHEN THE CORE IS EXECUTING SOME PARTICULAR THREADS IN SECURE USER MODE. PARTICULAR HARDWARE MUST BE DEDICATED FOR THIS, OR RE-USE BREAKPOINT REGISTER PAIR: CONTEXT ID MATCH MUST ENABLE TRACE INSTEAD OF ENTERING DEBUG STATE

FIG. 69B

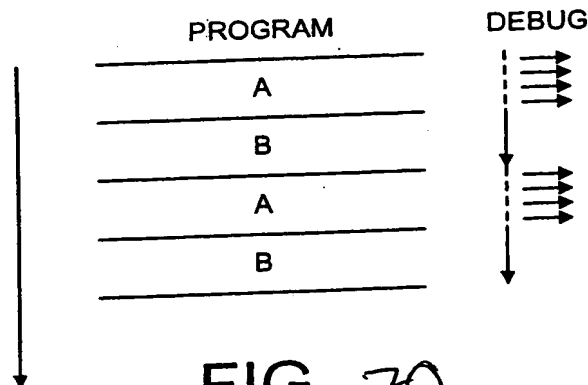


FIG. 70

62/62

METHOD OF ENTRY	ENTRY WHEN IN NON-SECURE WORLD	ENTRY WHEN IN SECURE WORLD
BREAKPOINT HITS	NON-SECURE PREFETCH ABORT HANDLER	SECURE PREFETCH ABORT HANDLER
SOFTWARE BREAKPOINT INSTRUCTION	NON-SECURE PREFETCH ABORT HANDLER	SECURE PREFETCH ABORT HANDLER
VECTOR TRAP BREAKPOINT	DISABLED FOR NON-SECURE DATA ABORT AND NON-SECURE PREFETCH ABORT INTERRUPTIONS. FOR OTHER NON-SECURE EXCEPTIONS, PREFETCH ABORT	DISABLED FOR SECURE DATA ABORT AND SECURE PREFETCH ABORT EXCEPTIONS <sup>(1)</sup> . FOR OTHER EXCEPTIONS, SECURE PREFETCH ABORT
WATCHPOINT HITS	NON-SECURE PREFETCH ABORT HANDLER	SECURE DATA ABORT HANDLER
INTERNAL DEBUG REQUEST	DEBUG STATE IN HALT MODE	DEBUG STATE IN HALT MODE
EXTERNAL DEBUG REQUEST	DEBUG STATE IN HALT MODE	DEBUG STATE IN HALT MODE

<sup>(1)</sup> SEE INFORMATION ON VECTOR TRAP REGISTER

<sup>(2)</sup> NOTE THAT WHEN EXTERNAL OR INTERNAL DEBUG REQUEST IS ASSERTED, THE CORE ENTERS HALT MODE AND NOT MONITOR MODE

FIG. 71A

METHOD OF ENTRY	ENTRY WHEN IN NON-SECURE WORLD	ENTRY WHEN IN SECURE WORLD
BREAKPOINT HITS	NON-SECURE PREFETCH ABORT HANDLER	BREAKPOINT IGNORED
SOFTWARE BREAKPOINT INSTRUCTION	NON-SECURE PREFETCH ABORT HANDLER	INSTRUCTION IGNORED <sup>(1)</sup>
VECTOR TRAP BREAKPOINT	DISABLED FOR NON-SECURE DATA ABORT AND NON-SECURE PREFETCH ABORT INTERRUPTIONS. FOR OTHER INTERRUPTION NON-SECURE PREFETCH ABORT	BREAKPOINT IGNORED
WATCHPOINT HITS	NON-SECURE DATA ABORT HANDLER	WATCHPOINT IGNORED
INTERNAL DEBUG REQUEST	DEBUG STATE IN HALT MODE	REQUEST IGNORED
EXTERNAL DEBUG REQUEST	DEBUG STATE IN HALT MODE	REQUEST IGNORED
DEBUG RE-ENTRY FROM SYSTEM SPEED ACCESS	NOT APPLICABLE	NOT APPLICABLE

<sup>(1)</sup> AS SUBSTITUTION OF BKPT INSTRUCTION IN SECURE WORLD FROM NON-SECURE WORLD IS NOT POSSIBLE, NON-SECURE ABORT MUST HANDLE THE VIOLATION.

FIG. 71B